EU-Rent Car Rentals Specification

Leonor Frias, Anna Queralt[1] and Antoni Olivé[2]
(*leonor.frias@estudiant.upc.es, [aqueralt | olive]@lsi.upc.es*)

1. INTRODUCTION AND MOTIVATION

EU-Rent is a widely known case study being promoted as a basis for demonstration of product capabilities. However, no in-deep case analysis neither specification had been developed. As far as we are concerned, all available documents only referred to a part of the system and did not confront some definition holes or even ambiguities of the case.

Therefore it was considered interesting, useful and even necessary to develop an in-depth study of the case which would lead to its whole specification. Having a complete specification of the case should be useful for its *users*, which could have a common reference.

On the other hand, it was considered a good opportunity to test the *real* application of some proposals. The first group consists on alternate mechanisms to define integrity constraints and derived elements proposed in [IC-Ol03][DR-Ol03] by Antoni Olivé. The root of these proposals is the definition of constraint and derived elements by means of operations.

Secondly, it was aimed to proof the utility and easy-to-use of an alternative approach of modelling events in which is still working Antoni Olivé. This alternative consists basically on modelling the events as objects and so, exploit the advantages of the OO.

Furthermore, although it was not an initial objective, the specification of this case has been useful to experiment with a few of the latest releases introduced in UML 2.0 and OCL 2.0 as some of these new mechanisms were needed (or practical) for the project working out.

This document is structured as follows: firstly the original case study is reviewed as some extensions are introduced, then, general remarks about the specification (language and tools used) are made, and next, the complete specification is presented with some previous explanations in each section. Lastly, conclusions and success of the overall work are commented on.

[1] Reviewer
[2] Director

2. THE CASE STUDY: EU-Rent Car Rentals

Overview

EU-Rent is a case study being promoted as a basis for demonstration of product capabilities which originally was developed by Model Systems, Ltd. It presents a car rental company with branches in several countries which provides typical rental services. Apart from collecting information about cars, branches…etc, effort is done to capture information about customers (if they are good clients or had had bad experiences otherwise).

Firstly, we will present the original case study, and then, some extensions widely used about pricing and discounting. These extensions were developed by Inastrol, and have been of great importance for all the interesting business rules associated in determining the price of a rental agreement. Documents from [BRF03] and [EBRC03] have been used for this section.

Lastly, we will expose some clarifications of some aspects of the case being judged obscure. The decisions made are intended to be consistent with the rest of the case and refer to other documents ideas when possible. Apart from own ideas, suggestions from [BRF03] , [EBRC03], [PSZ00] have been used for that section.

The original case

EU-Rent is a car rental company owned by EU-Corporation. It is one of three businesses - the other two being hotels and an airline - that each has its own business and IT systems, but with a shared customer base. Many of the car rental customers also fly with EU-Fly and stay at EU-Stay hotels.

EU-RENT BUSINESS

EU-Rent has 1000 branches in towns in several countries. At each branch cars, classified by car group, are available for rental. Each branch has a manager and booking clerks who handle rentals.

Rentals

Most rentals are by advance reservation; the rental period and the car group are specified at the time of reservation. EU-Rent will also accept immediate ("walk-in") rentals, if cars are available.

At the end of each day cars are assigned to reservations for the following day. If more cars have been requested than are available in a group at a branch, the branch manager may ask other branches if they have cars they can transfer to him.

Returns

Cars rented from one branch of EU-Rent may be returned to a different branch. The renting branch must ensure that the car has been returned to some branch at the end of the rental period. If a car is returned to a branch other than the one that rented it, ownership of the car is assigned to the new branch.

Servicing

EU-Rent also has service depots, each serving several branches. Cars may be booked for maintenance at any time provided that the service depot has capacity on the day in question.

For simplicity, only one booking per car per day is allowed. A rental or service may cover several days.

Customers

A customer can have several reservations but only one car rented at a time. EU-Rent keeps records of customers, their rentals and bad experiences such as late return, problems with payment and damage to cars. This information is used to decide whether to approve a rental.

EU-RENT BUSINESS RULES

External constraints

- Each driver authorized to drive the car during a rental must have a valid driver's licence.
- Each driver authorized to drive the car during a rental must be insured to the level required by the law of each country that may be visited during the rental.
- Rented cars must meet local legal requirements for mechanical condition and emissions for each country that may be visited during the rental.
- Local tax must be collected (at the drop-off location) on the rental charge.

Rental reservation acceptance

- If a rental request does not specify a particular car group or model, the default is group A (the lowest-cost group).
- Reservations may be accepted only up to the capacity of the pick-up branch on the pick-up day.
- If the customer requesting the rental has been blacklisted, the rental must be refused.
- A customer may have multiple future reservations, but may have only one car at any time.

Car allocation for advance reservations

At the end of each working day, cars are allocated to rental requests due for pick-up the following working day. The basic rules are applied within a branch:

- Only cars that are physically present in EU-Rent branches may be assigned.
- If a specific model has been requested, a car of that model should be assigned if one is available. Otherwise, a car in the same group as the requested model should be assigned
- If no specific model has been requested, any car in the requested group may be assigned
- The end date of the rental must be before any scheduled booking of the assigned car for maintenance or transfer
- After all assignments within a group have been made, 10% of the group quota for the branch (or all the remaining cars in the group, whichever number is lower) must be reserved for the next day's walk-in rentals. Surplus capacity may be used for upgrades.

- If there are not sufficient cars in a group to meet demand, a one-group free upgrade may be given (i.e. a car of the next higher group may be assigned at the same rental rate) if there is capacity
- Customers in the loyalty incentive scheme have priority for free upgrades.

If demand cannot be satisfied within a branch under the basic rules, one of the 'exception' options may be selected:

- A car may be allocated from the capacity reserved for the next day's walk-ins.
- A 'bumped upgrade' may be made. *(For example, if a group A car is needed and there is no capacity in group A or B, then a car allocated to a group B reservation may be replaced by a group C car, and the freed-up group B car allocated to the group A reservation.)*
- A downgrade may be made.
    - A "downgrade" is a car of a lower group.
- A car from another branch may be allocated, if there is a suitable car available and there is time to transfer it to the pick-up branch.
- A car due for return the next day may be allocated, if there will be time to prepare it for rental before the scheduled pick-up time.
- A car scheduled for service may be used, provided that the rental would not take the mileage more than 10% over the normal mileage for service.

If demand cannot be satisfied within a branch under the 'exception' rules, one of the 'in extremis' options may be selected:

- Pick-up may have to be delayed until a car is returned and prepared.
- A car may have to be rented from a competitor.

Walk-in rentals

- The end date of the rental must be before any scheduled booking of the assigned car for maintenance or transfer.
- If there are several available cars of the model or group requested, the one with the lowest mileage should be allocated.

Handover

- Each driver authorized to drive the car during a rental must be over 25 and have held a driver's license for at least one year.
- The credit card used to guarantee a rental must belong to one of the authorized drivers; and this driver must sign the rental contract. Other drivers must sign an 'additional drivers authorization' form.
- The driver who signs the rental agreement must not currently have a EU-Rent car on rental.
- Before releasing the car, a credit reservation equivalent to the estimated rental cost must be made against the guaranteeing credit card.
- The car must not be handed over to a driver who appears to be under the influence of alcohol or drugs.
- The driver must be physically able to drive the car safely - must not be too tall, too short or too fat; if disabled, must be able to operate the controls.
- The car must have been prepared -- cleaned, full tank of fuel, oil and water topped up, tires properly inflated.
- The car must have been checked for roadworthiness -- tire tread depth, brake pedal and hand brake lever travel, lights, exhaust leaks, windscreen wipers.

No-shows

- If an assigned car has not been picked up 90 minutes after the scheduled pick-up time, it may be released for walk-in rental, unless the rental has been guaranteed by credit card.
- If a rental has been guaranteed by credit card and the car has not been picked up by the end of the scheduled pick-up day, one day's rental is charged to the credit card and the car is released for use the following day.

Return from rental

- At the end of a rental, the customer may pay by cash, or by a credit card other than the one used to guarantee the rental.
- If a car is returned to a location other than the agreed drop-off branch, a drop-off penalty is charged.
- The car must be checked for wear (brakes, lights, tires, exhaust, wipers etc.) and damage, and repairs scheduled if necessary.
- If the car has been damaged during the rental and the customer is liable, the customer's credit card company must be notified of a pending charge.

Early returns

- If a car is returned early, the rental charge is calculated at the rate appropriate to the actual period of rental (e.g. daily rate rather than weekly).

Late returns

- If the car is returned late, an hourly charge is made up to 6 hours' delay; after 6 hours a whole day is charged.
- A customer may request a rental extension by phone -- the extension should be granted unless the car is scheduled for maintenance.
- If a car is not returned from rental by the end of the scheduled drop-off day and the customer has not arranged an extension, the customer should be contacted.
- If a car is three days overdue and the customer has not arranged an extension, insurance cover lapses and the police must be informed.

Car maintenance & repairs

- Each car must be serviced every three months or 10,000 kilometres, whichever occurs first.
- If there is a shortage of cars for rental, routine maintenance may be delayed by up to 10% of the time or distance interval (whichever was the basis for scheduling maintenance) to meet rental demand.
- Cars needing repairs (other than minor body scratches and dents) must not be used for rentals.

Car purchase and sale

- Only cars on the authorized list can be purchased.
- Cars are to be sold when they reach one year old or 40,000 kilometres, whichever occurs first.

Car ownership

- A branch cannot refuse to accept a drop-off of a EU-Rent car, even if a one-way rental has not been authorised.

- When a car is dropped off at a branch other than the pick-up branch, the car's ownership (and, hence, responsibility for it) switches to the drop-off branch when the car is dropped off.
- When a transfer of a car is arranged between branches, the car's ownership switches to the 'receiving' branch when the car is picked up.
- In each car group, if a branch accumulates cars to take it more than 10% over its quota, it must reduce the number back to within 10% of quota by transferring cars to other branches or selling some cars.
- In each car group, if a branch loses cars to take it more than 10% below its quota, it must increase the number back to within 10% of quota by transferring cars from other branches or buying some cars.

Loyalty incentive scheme

- To join the loyalty incentive scheme, a customer must have made 4 rentals within a year.
- Each paid rental in the scheme (including the 4 qualifying rentals) earns points that may be used to buy 'free rentals.'
- Only the basic rental cost of a free rental can be bought with points. Extras, such as insurance, fuel and taxes must be paid by cash or credit card.
- A free rental must be booked at least fourteen days before the pick-up date.
- Free rentals do not earn points.
- Unused points expire three years after the end of the year in which they were earned.

EXAMPLES OF "RULES FOR RUNNING THE BUSINESS"

(not really the same kind of rules as those above)

- Each branch must be set targets for performance -- numbers of rentals, utilization of cars, turnover, profit, customer satisfaction, etc.
- Where performance requirements conflict (e.g. profit vs. customer satisfaction when a customer requests a reduction in charges after an unsatisfactory rental) heuristics must be provided to guide branch staff.
- Performance data must be captured.

If performance targets are not met, control action must be taken. Control action may include:

- changing the resources at branches (e.g. numbers of cars, quotas of cars within each group, number of staff),
- changing responsibilities (e.g. having transfers of cars managed by groups of branches, rather than by negotiation between individual branch managers),
- changing operational guidance (e.g. what proportion of cars should be kept for walk-in rentals), but not external constraints (e.g. legal requirements) or company policies (e.g. rentals must be guaranteed by a credit card, a customer may have only one car at a time).

Assumed extensions about pricing and discounting

Standard Price

Rental Duration Category provides the allowable set of rental durations. For each duration, the unit of measure is provided (e.g. week, day, hour) and the minimum and maximum limits for each duration. For example, a weekly rental is for a minimum of 6 days and a maximum of 7 days. EU-Rent doesn't have weekend rental durations.

A rental may cover multiple durations. For example, a 10-day rental consists of 1 weekly rental and 3 daily rentals.

While EU-Rent will rent cars on an hourly basis during one day, it won't rent for portions of a day. So, a customer can't request to rent a car for 3 days and 5 hours. But he could ask to rent a car for just 5 hours.

Car Group Duration Price provides the standard rates for each Car Group by Rental Duration Category.

To develop the standard price for a rental agreement

- Break down the total rental duration into duration categories (weeks, days etc.) and determine the number of units for each,
- For each duration category, select the duration price for the car group, multiply each price by the number of units.
- Add all the results to get the total price.

Discounting

A rental may qualify for discounts under a number of promotions, such as a Loyalty Program Member discount or a "week long" rental discount. Only one (the best one) is used to calculate the rental price.

A customer must always receive the best price for a rental, regardless of promotions that were in effect when they made their reservations. At each customer touch point (e.g. make a reservation, pick up the car, return the car) the pricing business rules are applied to determine whether the rental qualifies for a better discount. So, if a new promotion is put into place after the customer makes a reservation or even during the rental, the customer can benefit from that new promotion.

However, *provided that the rental duration is unchanged* the best price quoted is always honoured; e.g. if the best price at pick-up is higher than the price quoted at reservation, the reservation price is used, even if the rules and rates that applied at reservation time are no longer current.

If the rental duration is changed - by rescheduling before pick-up, by returning the car earlier or later than the scheduled date - the price must be recalculated, using the best discount for the new duration.

EU-Rent's current discounted promotion programmes are

Name	Car Groups	Durations	Discount	Business Rules
3-day Advance	All	All	10%	All rentals booked at least 3 days in advance qualify for a 10% discount.
Summer Week	Mid-sized, Full Sized, Luxury, Sport Utility, Minivan	Weekly	€50.00	Weekly renters of a qualifying car receive a €50 discount.
New Loyalty Member	Compact, Mid-sized, Full Sized	Daily, Weekly, Monthly	2 car group upgrades	New loyalty club members are eligible for a 2 level upgrade, subject to availability on their first rental after joining the programme.

Proposed clarifications

This section seek mainly two aims. Firstly, it pretends to describe more clearly the entities of the system and their attributes while making some hypothesis; secondly, it pretends to make concrete clarifications about the logic of the business.

ENTITIES

In the case study, we can identify the following entities:

- **Branch:**
  - Attributes:
  - Capacity (on a day): There is no definition in the context of the case, and from our point of view, the term *availability* would represent the concept more accurately. We understand that *branch capacity* refers to the total number of cars available to rent on a concrete day (that is, the sum of the number of estimated cars of each group), and so, better corresponding with the idea of availability.

    Our suggestion to calculate the capacity of a day x is to add the capacity of the previous day(x-1), to the cars expected to be returned that day(x), and finally, subtract the cars which are already reserved for that day(x).
  - Location: on the road, medium city, big city, airport
- **Cars:**
  - They are classified by car group.
  - Important attributes:
    - Model: a rental could specify a preferred model.
    - Mileage: is an indicator for the state of the car (if it is in need of service or not). It will be necessary the current mileage and mileage from last service.
    - Date of last service.
  - State:
    - Available: not being rented, assigned or need maintenance.
    - Assigned: assigned to one of today's reservations and awaiting pick-up.
    - In rent: assigned to an open rental agreement
    - Needs maintenance: one of the conditions to need maintenance has become true.
    - Maintenance scheduled
    - Repairs scheduled
    - To be sold: cars that have reached one year old or 40,000 kilometres, or the ones that the manager has decided to sell due to a surplus.
    - Sold: cars that no longer belong to EU-Rent
- **Car group:**
  - Classification scheme that groups car models, based on common features.
  - Each car group has a quota, which corresponds to the desirable number of cars of a concrete type. We assume that every branch has its quota for each car group.
  - We won't assume that the number of car groups is fixed, however this hypothesis is made in some interpretation in [PSZ00].
- **Car model:**
  - The name given by a car manufacturer to a category of car that it produces.
- **Rental agreement:**

- There are two types of rentals.
  - Advance reservations: it must be supplied the rental period (that is, pick-up time and day of drop-off), the drop-off branch and car group (otherwise default group is used). We assume that countries through which the customer is going to travel are also given.
    Other additional information that may be supplied: model of preference, credit card to guarantee the rental(*).Recording the moment in which the reservation is made will be also useful for defining a priority criteria to allocate cars.
  - Walk-in rentals: immediate, depend on the availability of cars

  (*) The original text is rather contradictory in this aspect and we have assumed that the customer must not necessarily guarantee the rental, which is the suggested option until the last section (where it is stated the contrary).
- States:
  - Reservation: a car has been requested for a specific date
  - Open Rental Agreement: a car has been picked up and the car has not been returned.
  - Closed Rental Agreement: the car has been picked up and returned
  - Cancelled: any of the handover requirements has not been met, the customer has not shown or the customer has decided to cancel the reservation.
    We will consider that if the customer decides to cancel the rental before the pick-up day, it's free of charge (it is the policy that seems to be suggested by the way no-shows are treated). Otherwise, if it was a guaranteed rental, one-day rental will be charged.
- Other attributes:
  - Payment type: cash, credit card, loyalty club points
  - Base rental price: price before any discounts have been applied
  - Lowest price: lowest price offered since reservation
  - Actual return date and time

- **People (of interest to EU-Rent):**
  - Categories:
    - Customer: someone who once had or has a reservation with EU-Rent
      - member of the loyalty incentive scheme.
      - non member
    - Non customer: has been an additional driver in at least one rental or has tried to do a rental.
  - Historical information about customers is kept to decide whether to approve a rental. This information is a composite of rentals, bad experiences (such as late returns, problems with payment and damage to cars) and others.
  - They must have a valid driving license.
  - They must be over 25 (date of birth should be recorded).
  - Historical information about additional drivers should be recorded as well, in case they become customers in the future or to accept them as additional drivers again. This information includes: bad experiences, age, driving license.

- **Rental duration category:**
  - The minimum and maximum duration should be stored for each

- **Countries**:
  - Requirements for mechanical condition and emissions of cars should be recorded.
  - Other attributes: car renting tax

EU-RENT BUSINESS RULES

- **Rental reservation acceptance:**
  - Some extensions of EU-Rent use criteria to blacklist a person. In this aspect, we think the criteria should also take into account faults seriousness. Consequently, it will be necessary to describe more accurately the bad experiences to identify the degree of the fault, and have a concrete criteria to blacklist (because managers are not expected to examine client records). In the examples given we suggest:
    - late return: number of days/hours, justifying cause
    - problems with payment: amount of money, delay time
    - car damage: repairing cost (in days, hours).

    Intervals can be made to fix the degree of the fault.

    Additionally, we will assume that as soon as the blacklisting criteria are met (after a return, …), the customer will be blacklisted and all his reservations cancelled.

    Finally, we will assume for simplicity that once a person has been blacklisted, he or she will always remain blacklisted, since neither the original case nor any of the extensions specify the opposite. However, in a real case, this option should be included (to correct errors, misunderstandings, special situations…).

- **Car allocation for advance reservations:**

  A relevant fact that is not mentioned in the original case is the order in which car allocation is made, which is specially important when a reservation cannot be eventually accomplished. One reasonable criteria to solve this problem is to give more priority to the "best customers" (that is, in the loyalty incentive scheme), and then to guaranteed rentals.
  Finally, more priority can be given to the earliest reservations.

  However, this cannot be inconsistent with the fact of giving free upgrades to the best customers, so it is needed a previous estimation of the number of free upgrades that are going to be made, and then assign them firstly to clients in the loyalty incentive scheme. Furthermore, among clients within a kind of assignment (e.g. Clients who asked for class B and are going to be assigned class B), the preference criteria (model) will be applied to decide the concrete assignment.

  Lastly, we consider that in case the client is served after the day expected, he/she should be compensated, in the same way that he is required to pay a fine when has a bad experience. Additionally, an apologising letter will be sent as is suggested in an interpretation in [PSZ00] as we want to harm the least possible customer satisfaction.

  We leave the order in which exception criteria should be applied open to the user or the designer.

  Additionally, some remarks on the following exception rules:
  - *A downgrade may be made.*

    If this option is chosen, rental price will be calculated on the basis of the downgrade car group, instead of doing it with the desired car group as usual. This is done to prevent the customer from paying for more than what he is offered.

  - *A car from another branch may be allocated, if there is a suitable car available and there is time to transfer it to the pick-up branch.*

    First of all, we will assume that a suitable car is a car of the group demanded by the customer. Secondly, this alternative presents the following problem: if every branch can transfer to every branch, which first? With which criteria? As we have the

transportation time constraint, it seems reasonable that each branch can only transfer to a subset of all branches (for distance limitations or size of the branch). To have a concrete criteria to look for a transfer, we will assume that we know the expected time of transportation between related branches, and that minimum time is desirable. We will not assume bidirectionality of the relation (That is, A can transfer to B, but B not necessarily transfers to A).
Additionally, we will assume that ownership of the car is not effectively transferred until the car arrives to its destination.

- *A car due for return the next day may be allocated, if there will be time to prepare it for rental before the scheduled pick-up time*

This option implies 2 facts: firstly, it implies knowing that a car is going to be dropped off in that branch, and secondly, having an estimation of time needed to prepare a car to rent. We will assume that this time is composed by a basic time, needed for a simple inspection, and an additional time needed depending on the characteristics of the rental which is to be assigned.
To fulfil the first implication we will assume that the client *must* fix the branch of drop-off when doing the reservation. It is reasonable that this factor is fixed as it is stated in [EBRC03].

- **Walk-in rentals:**

It is not described what happens when no cars of the requested type are available. However, it is logical to think that available cars are going to be offered, paying in this case the fee corresponding to the rented car, not the corresponding to the initially desired car, as happens with reservations.

Furthermore, it is surprising that the mileage criteria is only used for walk-in rentals. However, we won't include it in reservations. We will lastly assume that mileage is referring to the absolute mileage, not relative to the last service.

- **Handover:**

First of all, if any of the handover conditions is not met, we assume that the reservation is cancelled.

Additionally, we consider that, in case that any of the conditions which the renter is responsible for is not fulfilled, he/she should be punished. Firstly, the renter should be aware of these conditions and consequently know his rights and duties. Secondly, in the case of late-returns and no-shows the customer is punished (at least economically), and so should happen in this case to be consistent.

We assume that if it was a reservation, the car assigned is freed and the renter is charged in an hourly base if it was a guaranteed rental (we adopt the policy used for no-shows). It also could be considered as a bad experience (as late-returns are), but we are not going to do it.

Furthermore, we consider similarly as it is suggested in [EBRC03], that additional drivers who have been blacklisted must be refused. This implies that additional drivers behaviour has to be judged as customers' is.

Lastly, we consider that additional drivers information(who, driving license..) is not known until the handover, as the rental procedure does not mention anything about it.

- **No shows:**

  If a rental has been guaranteed by credit card and the car has not been picked up by the end of the scheduled pick-up day, one day's rental is charged to the credit card and the car is released for use the following date.

  To ease the assignment of cars to reservations, we will assume that cars can only be picked up or returned during what is considered the working day. Otherwise, we could not be able to use cars (for new assignments) whose driver has not appeared. Besides, a 24-hour service is not suggested in the case, so we consider it a reasonable hypothesis (as happens with real car rent companies).

- **Return from rental:**

  We assume that car checking will be done as soon as possible (not more than a working day unless big repairs are needed) and in the same branch. We suppose that every branch is equipped with a garage service, where checking and repairs are done. Whereas, the responsibilities of service depots seem to be only periodic maintenance, not checking before and after a rental. Repairs and maintenance can't overlap.

  In case any damage has been recorded, it will be taken into account for all drivers. However, we will consider that the other bad experiences (problems with payment, late return…) will be only taken into account for the renter.

- **Late returns**:

  One additional problem that late returns (without arranged extension) imply refers to possible current assignments of cars returned late. A car could be assigned for the same day if the corresponding exception rule has been applied. In that case, the only reasonable solution would be delaying the handover unless there is an available car. In that case, the unsatisfied renter should be compensated as stated previously in the document.

  Furthermore, extensions should only be allowed until 1 day before the agreed date of return, otherwise the same problem as with late returns (without arranged extension) may happen. If this is fulfilled and provided there are not overlaps with other reservations of the same customer, we assume that there is no limit in the number of extensions.

- **Car maintenance & repairs:**

  We will consider that although initial date of car maintenance can be delayed as it is stated, we will achieve this effect by cancelling and rescheduling it to avoid more problems. Additionally, we will assume that both car maintenance and repairs must last for the expected duration (there will not be possible extensions). This conservative decision is taken because no major attention has been paid to this aspect in the bibliography, and it has been considered top priority to keep the simplicity of the case.

- **Car purchase and sale:**

  The original case states that only cars on the authorized list can be purchased. We will assume that the authorized list includes car models which are to offer in EU-Rent, and their corresponding car groups.

  However, it is not described how this process works. We will assume consequently a simplified process consisting of ordering a car and receiving a car (moment which is acquired effective ownership, although since its ordered it's owned with the object of accounting).

- **Car ownership:**

  In the case is stated that responsibility for a car is switched when car is dropped off, consequently, while the car is being rented, the renter branch has its ownership. If the car is rented from a competitor, we consider that no branch has ownership of it, but responsibility about it.

  Then, in order to manage the difference between the desirable quota of a car group and the actual number of car it owns (that includes the ones being currently rented by the stated before), both car transfers and selling are possible options. However, is not specified a preferable option or to which branches cars can be transferred.

  It is logical that the preferable option is transferring cars. However, transferring should not collapse the target branches (if there is surplus) or leave a branch with not enough cars of a car group. So it may be necessary to transfer cars to different locations. We suggest that cars can be transferred from one branch to another if there is a transfer agreement, and the transferring branch has excess, the receiver has less than its quota of that type of car and transferring does not exceed that quota. Finally, if there is still surplus, oldest cars of that group will be sold, while if there were not enough cars, cars should be bought.

- **Loyalty incentive scheme**:

  First of all, it is not clear from the case whether a customer automatically becomes a member of the loyalty incentive scheme when the criteria is achieved, or on the other hand, some action should be taken. However, the use of the verb 'join' suggests that the client should make some specific action. It is also the hypothesis that follow [BRF03].

  Besides, we will assume that no quota should be paid for becoming or being a member of the loyalty member scheme. In addition, we assume that once a customer has become a member, he or she will be so until he/she declines (previous reservations will not lose the discounts), makes no rental within a year or records a bad experience.

  With regard to the requirements to become a member of the loyalty incentive scheme, it is stated that a customer must have made 4 rentals within a year. However, it is natural to think that if a bad experience is recorded, points or accumulated rentals are lost, as this scheme promotes *good* customers.

  Furthermore, it is not fixed how points are assigned. We will assume that points are given upon the cost of rental (for example each x euros, a point is given) and only if the rental has not been qualified as a bad experience (due to reasons defended before).

  Another interesting point is the policy referring to how points can be spent. It is exposed that extras such as insurance, fuel and taxes must be paid by cash or credit card. However, this description of extras is a bit misleading. First of all, a prepared car must have the tank full of oil (so we do not consider that as an extra service), and secondly, insurance as is made reference in the case (in section about returns) seems to be referring to a company insurance, not customer's .

  Additionally, concerning points use, we will assume as in [EBRC03], that a base price for a rental (without taxes) must be entirely paid with points. We also take the same criteria of not benefiting then from discounts.

- **Pricing and discounting:**
  If the duration of a rental is changed previous discounts are not applicable, that is, final price will be calculated upon current or future fees. It is the only point in which payment type can (and should) be reconsidered (if loyalty points are used or not).

  We could also consider the possibility of adding new parameters of discounting, such as a senior citizen discount, as is suggested in [BRF03], which will be the same for all branches.
  Additionally, we assume that the best discount is automatically applied in every touch point, except for offers for a better service but same price, such as upgrades due to being a brand new member of the loyalty program, as it is assumed in [BRF03]. In that case, the customer may decline the upgrade (and consequently, lose that discount) and then, the rest of discounts are applicable.

  Lastly, we assume a logical (but not mentioned) rule concerning fees between car groups. Fees for a higher car group, in each of the durations, will be equal or more expensive than the lower groups.

EXAMPLES OF "RULES FOR RUNNING THE BUSINESS"

- We assume that targets for performance can be different from one branch type to another. However, some common indicators can be assumed (we will assume the ones given as an example).
- Some additional performance criteria suggested in bibliography, which we are not going to take into account, are:
  - In some interpretation of the case in [PSZ00] the most popular models within a branch are recorded. This may give priority to buy cars of popular models.
  - Track the discounts actually applied to rentals to analyse the effectiveness of the discount program. Suggested in [BRF03].
- Similarly, heuristics can change from one branch to another, as different countries can have different values. Another factor can be the size of the branch or location(if it is in a big city or not, in an airport…). This division is used in some cases of [PSZ00].
  Consequently, it can be assumed that each kind of branch within a country defines its additional indicators and heuristics.
- Finally, actions to be taken will be grouped as is stated in the case, and each kind of branch within a country can define the concrete actions.

ADDITIONAL REMARKS

- **Sharing of client database** with companies EU-Fly and EU-Stay:
  It is not clear in the case how the different companies share customer data. We will suppose in the text that although there is actually a shared data, the term *new client* refers to a new client for EU-Rent, but not necessarily for EU-Stay and EU-Fly. This should be taken into account when defining the related operations of the system.

- **Interaction with the system**:
  Some of the services that the system must provide, such as reservations, could be served either from a EU-Rent branch or from the Internet (we can easily imagine that a potential costumer could fill in a formulary by the Internet to make a reservation). However, this would imply that some checkings could not be done until the handover, and so, if basic customer requirements such as driving license and age were finally not met, there would be stored data of someone who can't be a customer. So, in order not to

add unnecessary complexity, we will assume that all transactions apart from extension of a rental agreement, will be done *in situ* in a EU-rent branch.

- **Customer management:**
  Most of the actions to be managed in EU-Rent are naturally assigned to a branch (reservations, assignments…). However, where is customer management (without direct customer involvement) done? (For example, who invites a customer to the loyalty incentive scheme?) As central management does not seem a solution because is a wide area company and there is so many data and sensibilities to be taken into account, we choose the following criteria: a customer *belongs to* the EU-Rent branch where he/she firstly had a contact with. His/her management (notifications…) will be done there.

- **Units of measurement**: We have assumed euros for money and decimal metrical system for distances, as it is in the original case, despite in Inastrol case study miles are used for distances.

- **Money available of each branch:** We will assume that EU-Rent is a profitable company and there is no problem of money to pay expenses (a new car, service depots…). Calculating the available money of a branch would require knowing many data which is not defined: sum of staff salaries (and so, how many staff), cost of maintenance…etc.

3. GENERAL COMMENTARIES ABOUT THE SPECIFICATION

Notation and Language

In order to define the conceptual schema of the case EU-Rent, UML was chosen as base modelling language, while OCL was chosen for the definition of constraints. Additionally, the variations and proposals mentioned in the introduction will be used when appropriate.

Concerning versions, initially were implicitly chosen current versions 1.5 of UML and OCL. However, during the development of this project, some mechanisms were needed which were not offered by the current versions, and therefore it was considered reasonable to use the new proposals of UML 2.0 and OCL 2.0 (still not definitive).

Concretely, UML 2.0 notation has been used for the elaboration of sequence diagrams and OCL 2.0 has been used for the definition of some complex restrictions which needed the concept of Tuple which is introduced in 2.0 version.

Furthermore, it has been considered convenient to use some standard XML types relating to time, as UML does not define any by default. XML schema provides a concrete semantic of these types and their operations and no additional complexity wanted to be added.

Finally, it should be noticed that some *global* predicates have been used for convenience, such as now(), today(),…. In these cases, the chosen name has tried to be self descriptive.

Tools

To easily construct the diagrams which make up the conceptual model, the Rational Rose 98 Case Tool was mainly used. The choice of this tool was motivated by a mere criteria of previous knowledge of the tool and it was considered enough for the aim of this project.

However, once the project was being developed it showed to be rather poor in some aspects such as the lack of representation of n-ary associations, which were solved with some *graphical tricks.*

Additionally, the sequence diagrams were drawn *manually* with Microsoft Word. There were not significantly better alternatives as UML 2.0 notation is still not official and so, it is not supported by case tools.

4. USE CASES

Notation

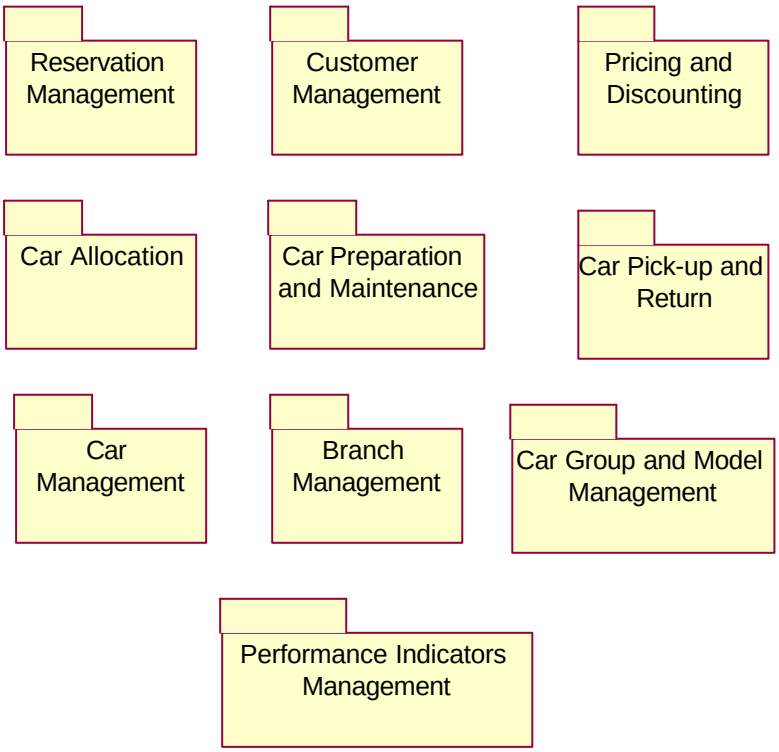
To describe the system use cases we have chosen a rather simple 2-column template (that is, without preconditions and postconditions, GUI requirements …etc) which we have considered enough for the aim of this project. Additionally, we have had the need to structure the use-cases, mainly using the *include* stereotype, as there is not a total agreement about the use of *extend* stereotype and generalization.

In order to describe alternate courses and have a clear semantic reference of them, we have mainly applied the ideas proposed in [Coc00] although we have also taken into account commentaries in [Lar02] and [Gel03].

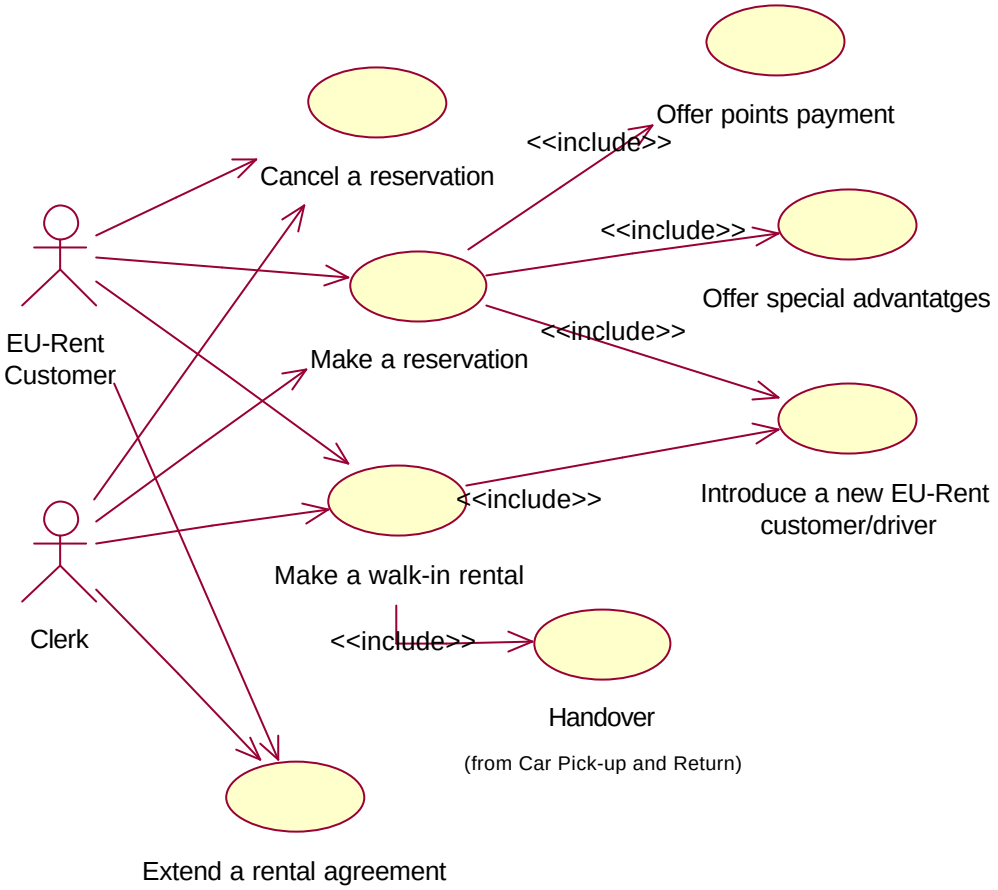
Finally, we have written the use cases which imply complex business rules with the aid of tables. This technique is suggested in [Wei03] to not clutter the schema.

Diagrams

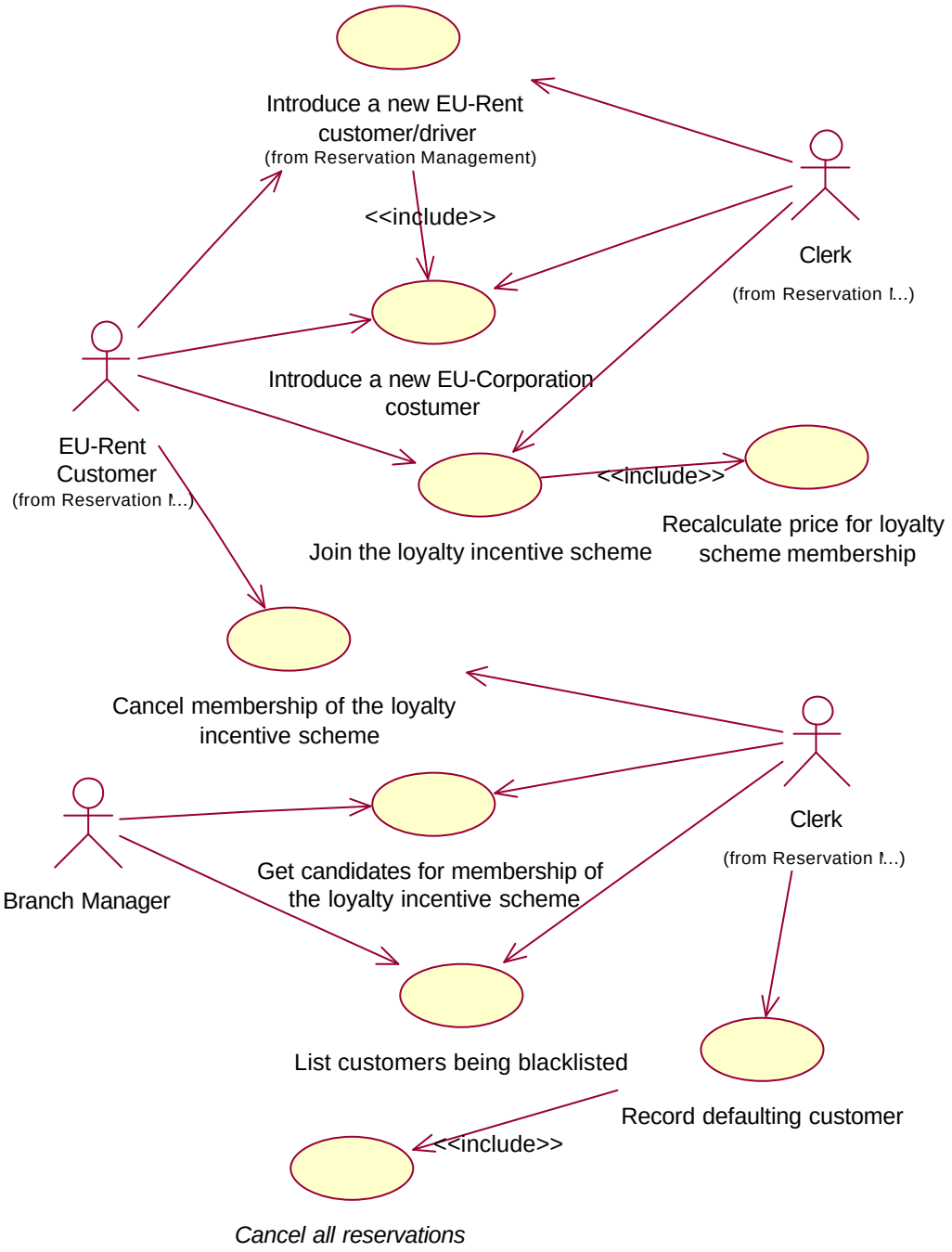
The modelling of use cases has been divided in packages, corresponding to thematic areas, in order to ease understanding. These are the following:



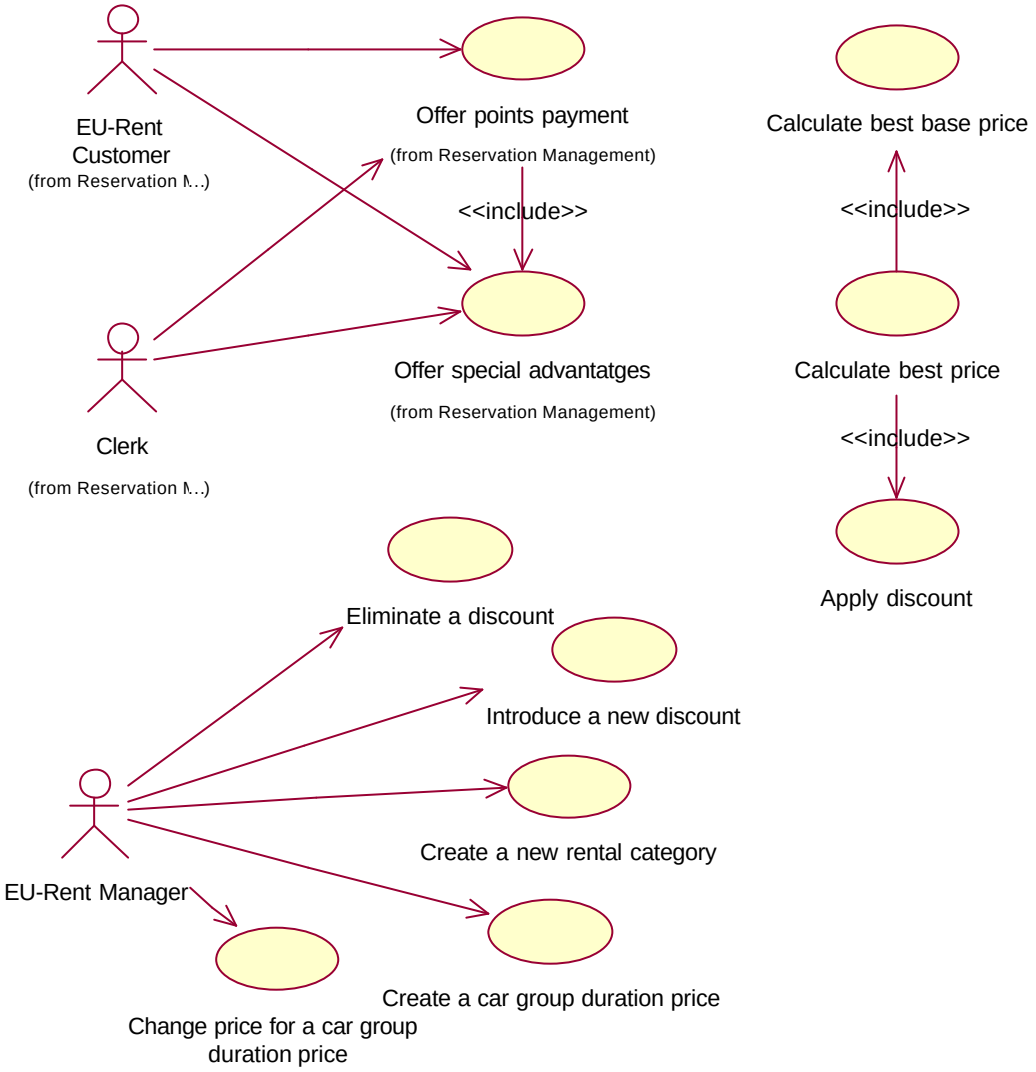
RESERVATION MANAGEMENT



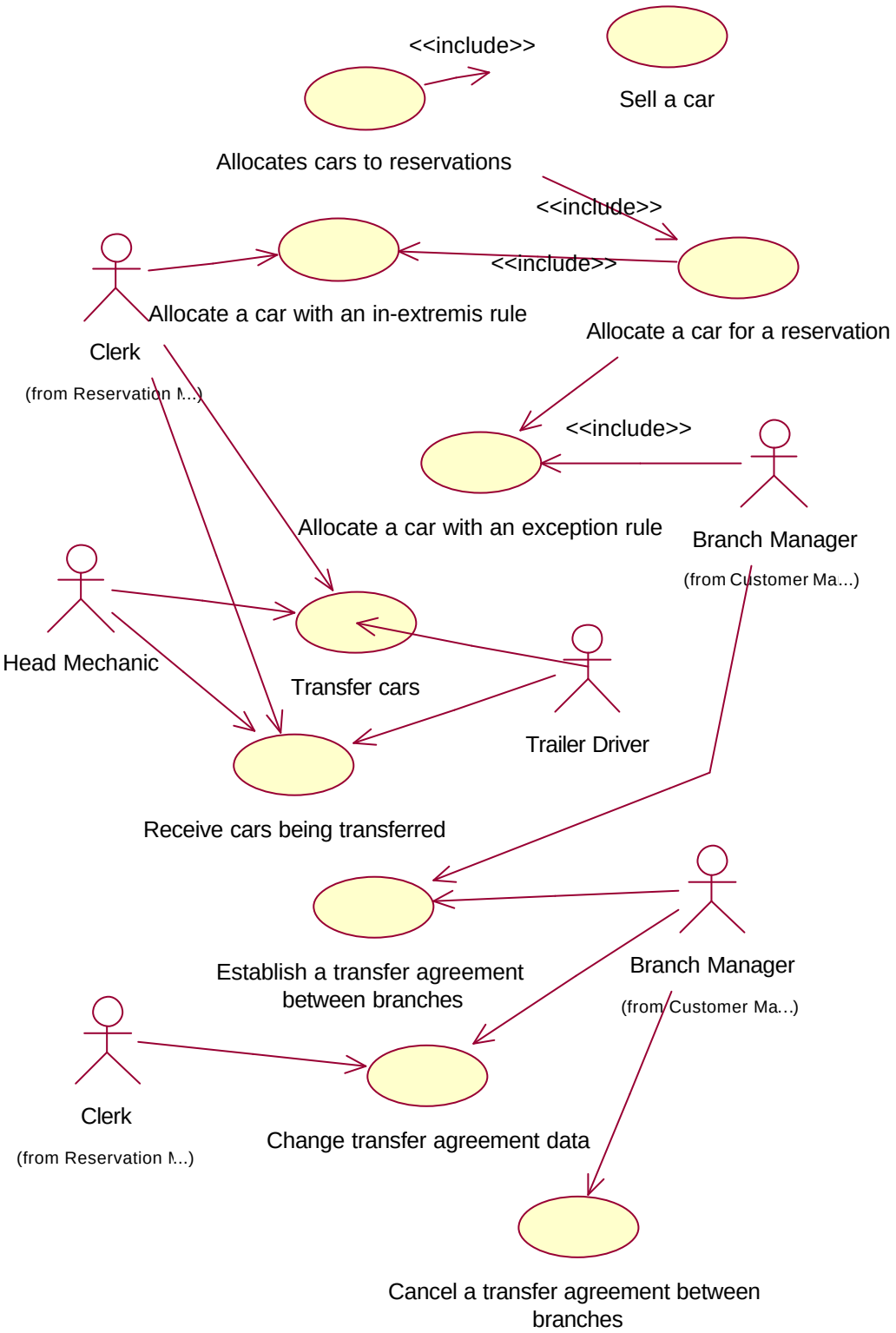
CUSTOMER MANAGEMENT



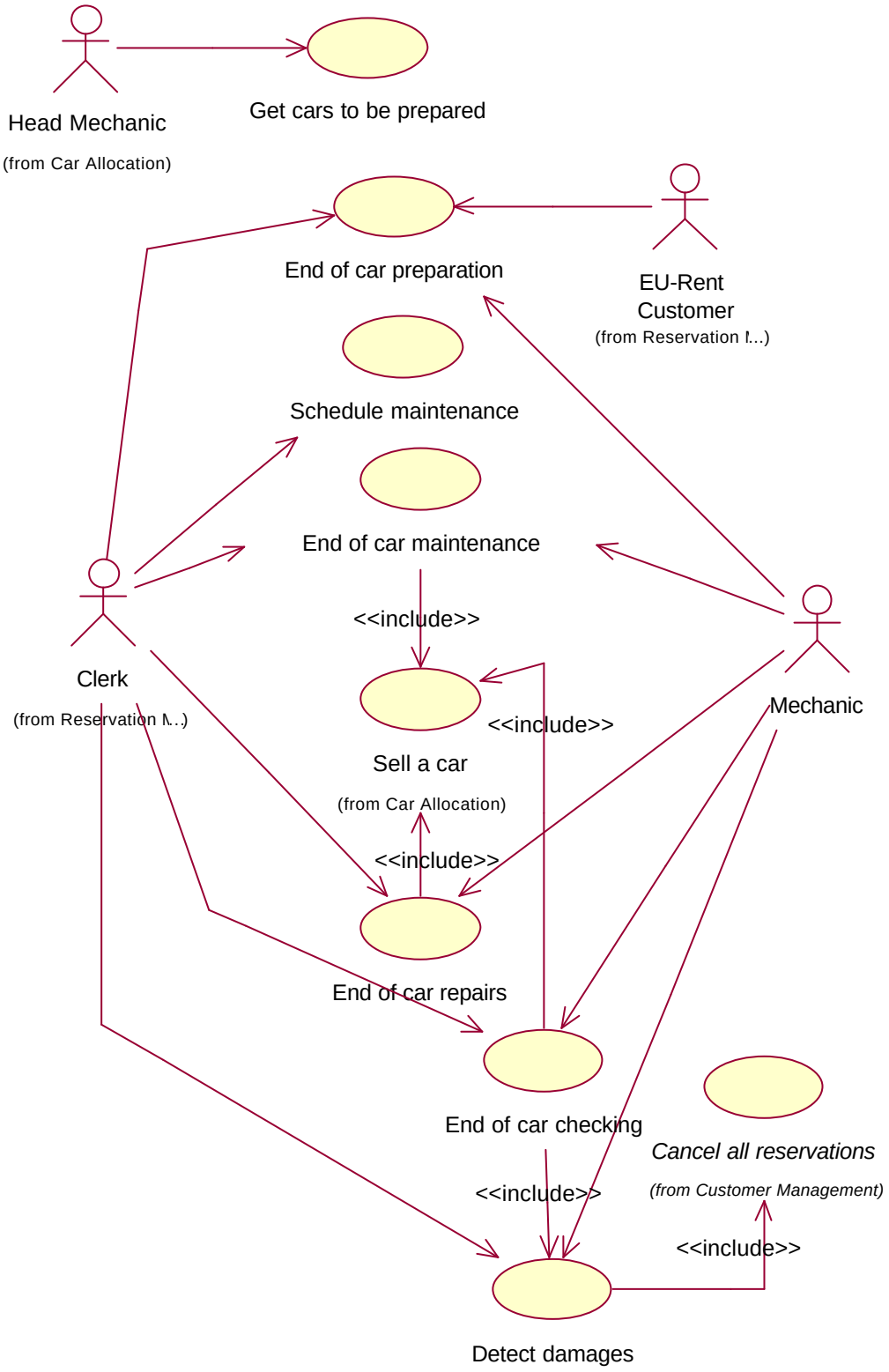
PRICING AND DISCOUNTING



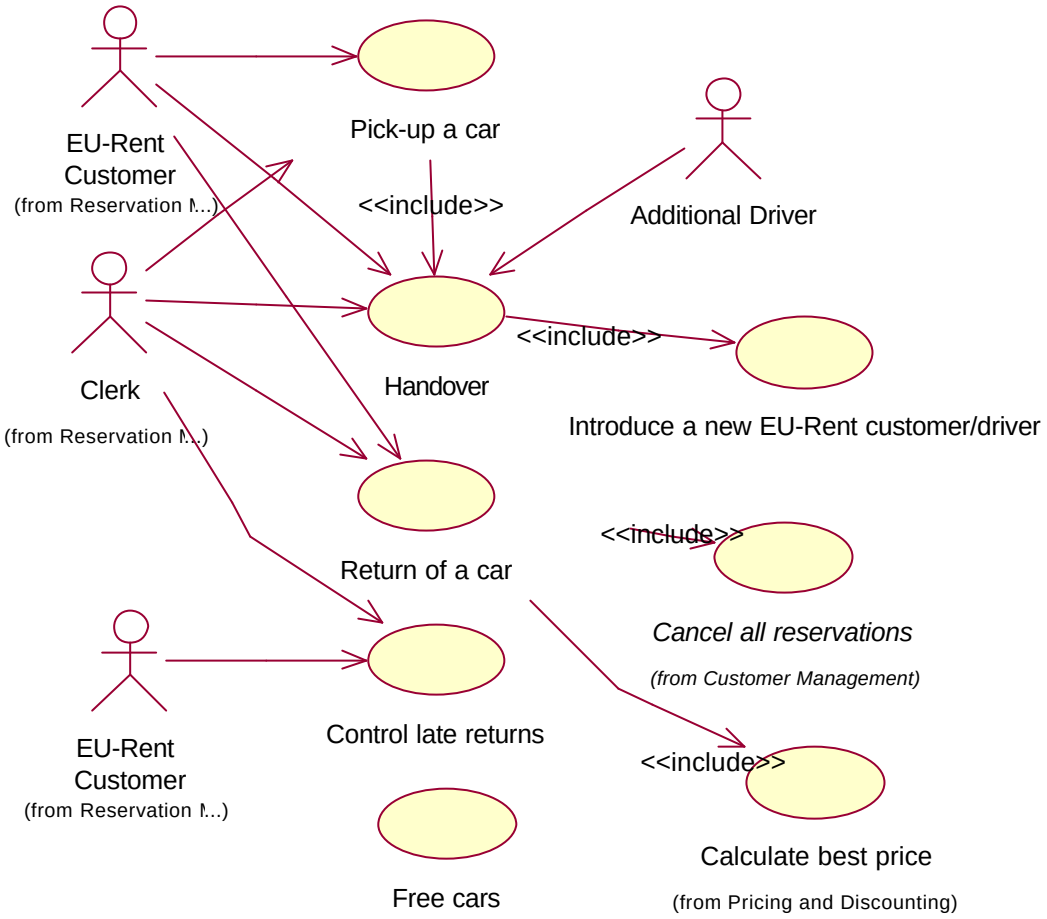
CAR ALLOCATION



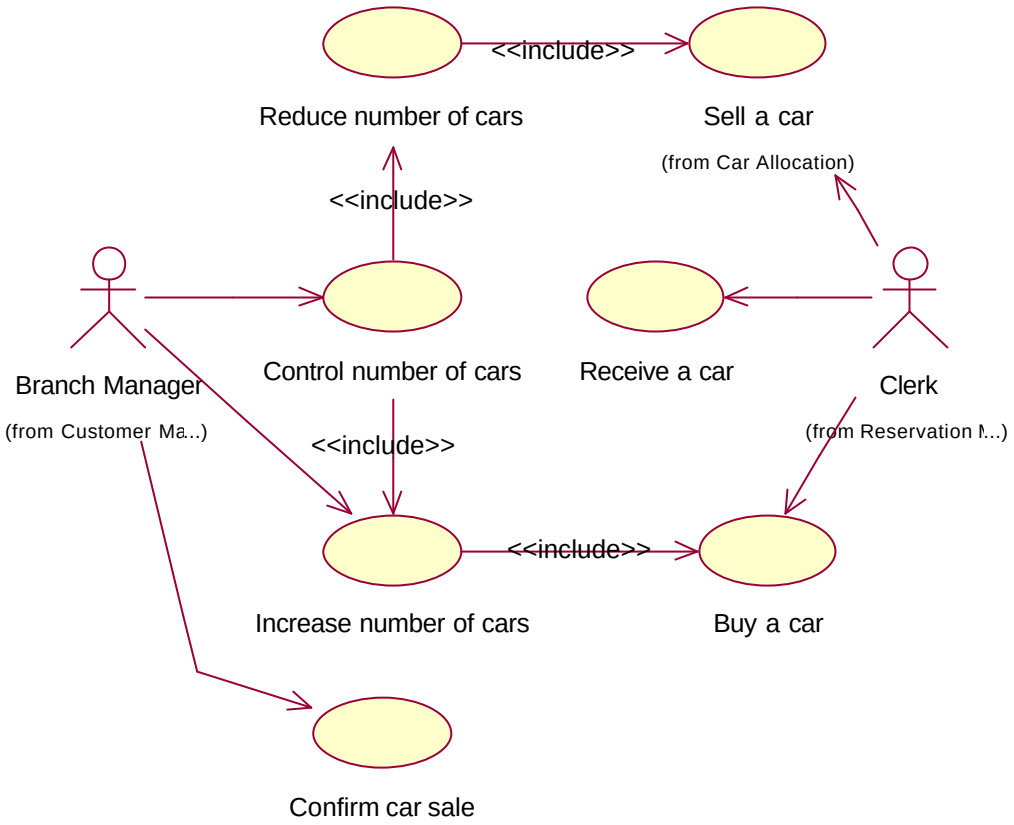
CAR PREPARATION AND MAINTENANCE



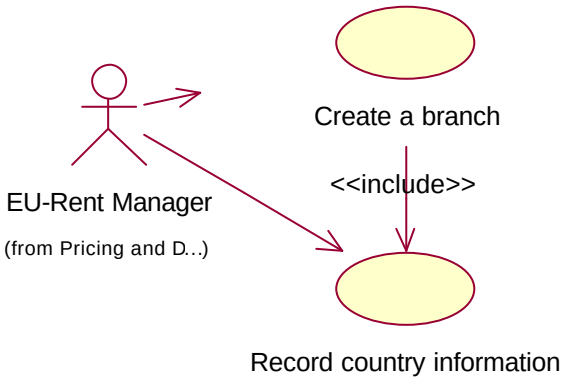
CAR PICK-UP AND RETURN



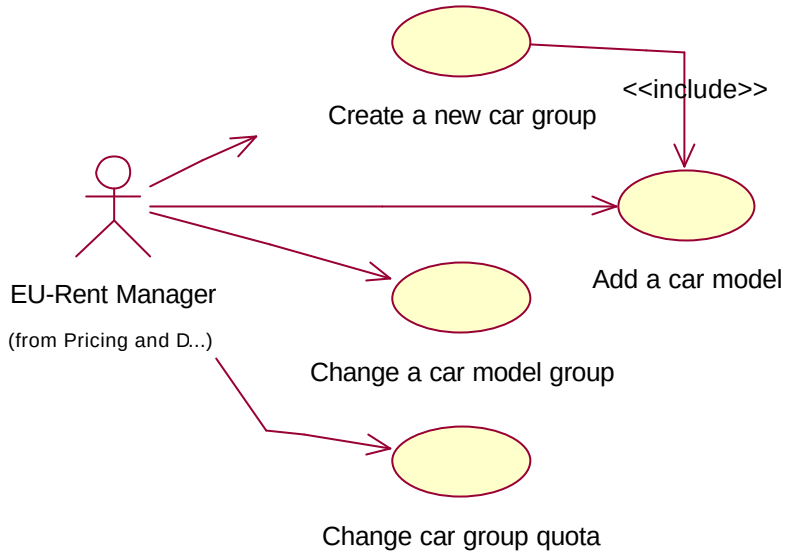
CAR MANAGEMENT



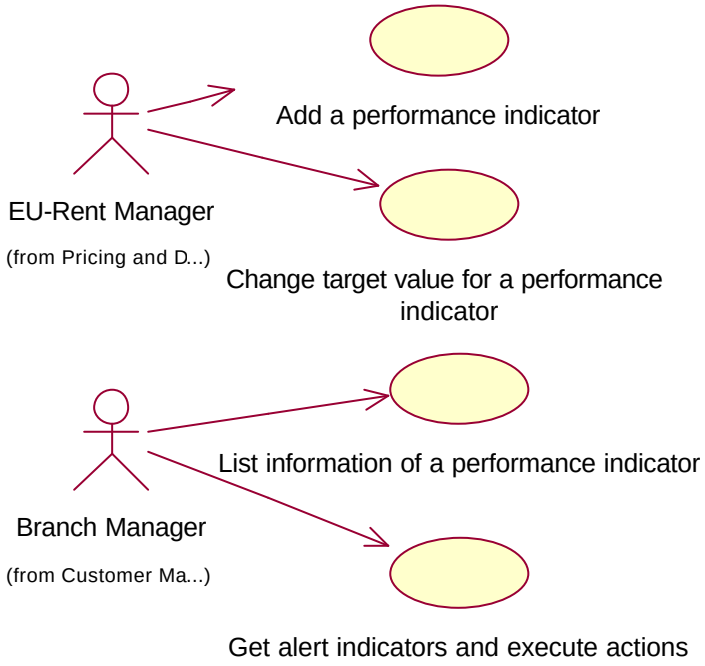
BRANCH MANAGEMENT



CAR GROUP AND MODEL MANAGEMENT



PERFORMANCE INDICATORS MANAGEMENT



Description

Reservation management

Use case: Make a reservation
Actors: Customer (initiator), Clerk
Overview: A customer makes a reservation from an EU-Rent branch
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a customer decides to make a reservation and tells it to an available clerk	
2. The clerk asks the costumer for his/her ID and introduces it.	
	3. Checks if the customer is a person who has had contact with EU-Rent. *a.* If he or she exists, verifies that the customer has not been blacklisted *b.* **Initiate** *Introduce a new EU-rent costumer/driver,* otherwise
4. The customer tells the information about the reservation to the clerk	
5. The clerk introduces the period desired, the pick-up branch, the drop-off branch, countries planned to visit and, optionally, the car group or the car model desired.	
	6. Verifies that the period is correct, that there is no overlap with other reservations of the customer and the availability of the specified car group or car model for the period indicated. If the customer has neither specified a car group nor a car model, he will be assigned a car belonging to the cheapest group. If the customer has specified a car model but there are no cars available of that model, a car of the same group will be assigned.
7. The clerk asks the customer if he/she wants to guarantee the rental. If so, a credit card number is introduced.	
	8. If a credit card number has been provided, the rental is guaranteed.
	9. Checks offers which must be selected a priori: *a.* If the customer is a member of the loyalty incentive scheme, **Initiate** *Offer points payment* b. **Initiate** *Offer special advantages,* otherwise

10. The rental is confirmed.

11. A new rental agreement is created with the indicated characteristics

SUCCESS EXIT

Extensions:

3a.a. The customer has been blacklisted: FAILURE EXIT.

6a. **-** The period is not correct:

- The period overlaps with other reservations of the customer:

Actor action	System responsibility
	6ª.1. Notifies the problem
6a.2. The clerk notifies the customer of the problem.	
6a.3. The customer either: a. Abandons the rental. SUCCESS EXIT. b. Specifies another period.	

6b. There is no availability of the specified car group or model for the complete period

Actor action	System responsibility
	6ª.1. Notifies the problem
6b.2. The clerk notifies the customer of the problem.	
6b.3. The customer either: c. Abandon the rental. SUCCESS EXIT. d. Specifies other data.	

10a. The rental is cancelled: SUCCESS EXIT

Use case: Extend a rental agreement
Actors: Customer (initiator), Clerk
Overview: A customer extends a rental agreement by phone
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a customer decides to extend his current rental agreement.	
2. The customer phones an EU-Rent Branch and tells his/her ID to the clerk who answers. The clerk introduces his/her ID and demands the extension.	
	3. Verifies if the extension is possible (no maintenance should be done).
4. The changes are confirmed.	
	5. The rental agreement is updated. SUCCESS EXIT

Extensions:

3a. Maintenance has been scheduled: FAILURE EXIT

Use case: Cancel a reservation by customer demand
Actors: Customer(initiator), Clerk
Overview: A customer decides to cancel a reservation
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a customer decides to cancel one of his/her reservations. He/She tells the clerk his/her ID and the beginning date of the reservation which wants to be cancelled.	
2. The clerk indicates the ID of the client, which one of his reservations wants to be cancelled, and the beginning date of the reservation which wants to be cancelled.	
	3. Verifies that a reservation for that day exists and has not been cancelled.
4. The cancellation is confirmed.	
	5. Looks for the pick-up day of the reservation. a. If the pick-up day is today and it was a guaranteed rental, the assigned car will be freed and 1 day-rental will be charged. Additionally, the system checks if there is a today reservation without car and the freed car can be assigned to it. b. The customer is charged with no costs, otherwise.
	6. The reservation is updated with cancelling details.
	SUCCESS EXIT

Extensions:
3a. Reservation doesn't exist:

Actor action	System responsibility
	2a.1. Notifies the problem
2a.2. The clerk notifies the customer of the problem.	
2a.3. The customer either: a. Abandons the cancellation. SUCCESS EXIT. b. Specifies another beginning date.	

3b. Reservation has already been cancelled: FAILURE EXIT

Use case: Make a walk-in rental
Actors: Customer (initiator), Clerk
Overview: A customer wants to make a walk-in rental
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a customer decides to make a walk-in rental and tells it to an available clerk	
2. The clerk asks the costumer for its ID and introduces it.	
	3. Checks if the customer is a person who has had contact with EU-Rent. a. If he or she exists, verifies that the customer has not been blacklisted b. **Initiate** *Introduce a new EU-rent costumer/driver,* otherwise
4. The customer tells the information about the reservation to the clerk	
5. The clerk introduces the period desired, the drop-off branch, countries planned to visit, the car group and a car model if customer specifies it.	
	6. Verifies that the period is correct, that there is no overlap with other reservations of the customer and that there is current availability (including non-guaranteed rentals whose renter has not shown after 90 min) of the specified car group. The system also verifies that the car is not scheduled for maintenance before the return date. If there are several cars available, the one with the lowest mileage should be allocated.
7. The rental is confirmed.	
	8. If the car was previously assigned to a no-show reservation, it is cancelled and the car is freed.
	9. A new rental agreement is created with the indicated characteristics
10. **Initiate** *Handover*	
11. The customer will wait until the car is prepared. SUCCESS EXIT	

Extensions:
3a.a. The customer has been blacklisted: FAILURE EXIT.

6a. **-** The period is not correct:
- The period overlaps with other reservations of the customer:

Actor action	System responsibility
	6a.1. Notifies the problem
6a.2. The clerk notifies the customer of the problem.	
6a.3. The customer either: a. Abandons the rental. SUCCESS EXIT. b. Specifies another period.	

6b. There is no availability of the specified car group on the current day

Actor action	System responsibility
	6b.1. Notifies the problem
6b.2. The clerk notifies the customer of the problem.	
6b.3. The customer either: c. Abandons the rental. SUCCESS EXIT. d. Specifies other data.	

10a. The rental is cancelled: SUCCESS EXIT

Customer management

Use case: Introduce a new EU-Rent customer/driver
Actors: Customer, Clerk
Overview: A new EU-Rent customer/driver is recorded.
Type: Abstract
Typical course of events:

Actor action	System responsibility
	1. Verifies that the customer is an existing customer of EU-Corporation.
	2. Otherwise, **initiate** *Introduce a new EU-Corporation customer.*
3. The clerk asks the customer for the driving license and verifies number, date of issue (at least 1 year of experience), date of expiration (it is still valid) .	
4. The clerk introduces the information.	
	5. Records the information of a new EU-rent customer belonging to the EU-Rent branch where the registration is done.
	SUCCESS EXIT

Extensions:
3a. The customer has less than a year of experience: FAILURE EXIT
3b. The driving license is not valid: FAILURE EXIT

Use case: Introduce a new EU-Corporation customer
Actors: Customer, Clerk
Overview: A new EU-Corporation customer is recorded.
Type: Abstract
Typical course of events:

Actor action	System responsibility
1. The clerk asks the customer for personal details (name, address, birthdate).	
2. The clerk introduces personal details	
	3. Records the information of a new EU-Corporation customer.
	SUCCESS EXIT

Extensions:
3a. The customer is below 25: FAILURE EXIT

Loyalty incentive scheme

Use case: Join the loyalty incentive scheme
Actors: Customer (initiator) ,Clerk
Overview: A *good* customer joins the loyalty incentive scheme
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a customer wants to join the loyalty incentive scheme.	
2. The clerk introduces customer ID.	
	3. Verifies that the customer is not already a member of the scheme.
	4. Verifies that the customer meets the requirements to become a member of the loyalty incentive scheme (he/she has made at least 4 rentals within a year and none of them has been qualified as a bad experience).
5. The membership is accepted.	
	6. The status of the client is changed and points are accumulated for the last 4 rentals.
	SUCCESS EXIT

Extensions:
3a. The customer is already a member of the scheme: FAILURE EXIT
4a. The customer doesn't meet the requirements: FAILURE EXIT

Use case: Cancel membership of the loyalty incentive scheme
Actors: Customer (initiator), Clerk
Overview: A current member of the loyalty incentive scheme declines membership
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a customer wants to decline membership of the loyalty incentive scheme.	
2. He/she tells to a clerk in a EU-Rent branch. The clerk introduces customer ID.	
	3. The status of the client is changed and points are lost.
	SUCCESS EXIT

Use case: Get candidates for membership of the loyalty incentive scheme
Actors: Clerk, Branch manager
Overview: Customers belonging to a branch which meet the requirements for being members of the loyalty incentive scheme are notified.
Type: Primary and essential, temporal
Typical course of events:

Actor action	System responsibility
1. At the end of each working day, an automatic process is initiated to get candidates for membership.	
	2. List all customers belonging to the branch which have met the requirements for being members of the loyalty incentive scheme during this day.
	3. Print the list of customers.
	4. Select one customer from the list and prepare a letter for him/her.
	5. Repeat 3 while there are more clients.
6. A clerk picks up the printout and the letters. He/she sends the letters and hands the printout to the branch manager.	
SUCCESS EXIT	

Blacklisting

Use case: List customers being blacklisted
Actors: Clerk, Branch manager
Overview: List candidates which are being blacklisted
Type: Primary and essential, temporal

Typical course of events:

Actor action	System responsibility
1. At the end of each working day, an automatic process is initiated to get candidates being blacklisted that day.	
	2. List all customers belonging to the branch which have met the requirements for being blacklisted (have recorded several bad experiences of certain seriousness) during this day.
	3. Print the list.
	4. Select one customer of the list and prepare a letter for him/her.
	5. Repeat 4 while there are more customers.
6. A clerk picks up the letters and sends them. He/she also picks up the printout and hand it to the branch manager	
SUCCESS EXIT	

Use case: Cancel all reservations
Actors: -
Overview: Cancel all reservations from a customer with the supplied motivation
Type: Abstract
Typical course of events:

Actor action	System responsibility
	1. Get all customer reservations.
	2. Select one reservation which has not been cancelled.
	3. The reservation is updated with supplied cancelling details.
	4. Repeat 2 and 3 while there are reservations which have been not cancelled
	SUCCESS EXIT

Payment problems

Use case: Record defaulting customer
Actors: Clerk (Initiator)
Overview: Record a customer's problem with payment
Type: Primary and essential

Typical course of events:

Actor action	System responsibility
1. The use case begins when a clerk receives a notification of a defaulting customer.	
2. The clerk introduces the customer's ID, the rental qualifying for bad experience and the seriousness of the problem.	
	3. Verifies that the customer and the rental exist.
	4. Records a bad experience for the customer of the indicated seriousness.
	5. If the customer was member of the Loyalty Incentive Scheme, he loses his membership.
	6. Checks criteria to be blacklisted a. If they are achieved, blacklist the person and **initiate** *Cancel all reservations* of the renter due to blacklisting SUCCESS EXIT

Extensions:
3a. The customer doesn't exist: FAILURE EXIT
3b. The rental doesn't exist: FAILURE EXIT

Pricing and discount management

Use case: Offer points payment
Actors: Customer ,Clerk
Overview: The price for a rental duration is calculated and the customer chooses points payment or discounts.
Type: Abstract

Typical course of events:

Actor action	System responsibility
	1. Verify that the reservation is being done or was done at least 14 days in advance.
	2. Calculate the current base price for the duration of the reservation for the desired car group.
	3. Verify that the customer has enough points
	4. Calculate applicable offers and show the best (best discount, free days,..) and price paying with points.
5. The clerk tells the customer which are the options and asks if he/she wants to pay with points.	
6. The clerk introduces customer's option	
	7. The rental will be paid in the chosen way.
	SUCCESS EXIT

Extensions:

1a. The reservation is not being done at least 14 days in advance:

Actor action	System responsibility
	1a.1. **Initiate** *Offer special advantages*
	SUCCESS EXIT

3a. The customer doesn't have enough points:

Actor action	System responsibility
	3a.1. **Initiate** *Offer special advantages*
	SUCCESS EXIT

Use case: Offer special advantages
Actors: Customer ,Clerk
Overview: The customer is offered some special advantages that must be decided at reservation time.
Type: Abstract
Typical course of events:

Actor action	System responsibility
	1. Select all special advantages currently applicable to the customer rental.
	2. Verify that there are special advantages applicable
	3. Show the options to the customer.
4. The clerk tells the customer which are the options and asks if he/she wants any of them or prefers normal discounts.	
5. The clerk introduces customer's decision result.	
	6. The rental will be paid in the chosen way.
	SUCCESS EXIT

Extensions:
2a. There are no special advantages applicable: SUCCESS EXIT

Use case: Calculate best base price
Actors: -
Overview: Calculate best base price for a customer rental
Type: Abstract
Typical course of events:

Actor action	System responsibility
	1. Check if the rental has been extended. a. If it has been so, obtain date of last extension. 2. Calculate the duration of the rental 3. Calculate best price for the rental duration and car group since reservation date or last extension (if it has been extended). 4. Show base price SUCCESS EXIT

Use case: Calculate best price
Actors: -
Overview: Calculate best price for a customer rental
Type: Abstract
Typical course of events:

Actor action	System responsibility
	1. **Initiate** *Calculate best base price*. 2. Check if points payment has been chosen: a. If it has been chosen, verify that the customer has enough points b. **Initiate** *Apply discount* with base price. 3. Show final price SUCCESS EXIT

Extensions:
2a. The customer has not enough points:

Actor action	System responsibility
	2a.1. Payment will not be done with points. 2a.2. **Initiate** *Apply discount* with base price

Use case: Apply discount
Actors: -
Overview: Calculate best discount for a customer rental
Type: Abstract

Typical course of events:

Actor action	System responsibility
	1. Calculate the applicable discounts on a customer's rental.
	2. For each applicable discount, calculate final price.
	3. Select best option.
	SUCCESS EXIT

Use case: Introduce a new discount
Actors: Manager (initiator)
Overview: Introduction of a new discount.
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a Manager from EU-Rent decides to offer a new discount.	
2. The manager introduces the name of the discount, the applicable car groups and durations, the concrete effect, a description, the date when it starts being available and if it is valid only for reservation time.	
	3. The information of the new discount is recorded.
	4. Each EU-rent branch is notified of the new discount
	SUCCESS EXIT

Extensions:
3a. The discount already exists: FAILURE EXIT

Use case: Eliminate a discount
Actors: Manager (initiator)
Overview: Eliminate a current discount.
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a Manager from EU-Rent decides to eliminate a current discount.	
2. The manager introduces the name of the discount.	
	3. Records that the discount is not applicable any more.
	SUCCESS EXIT

Extensions:
3a. The discount doesn't exist: FAILURE EXIT

Rental categories and their prices

Use case: Create a new rental duration
Actors: Manager(s) (initiator)
Overview: Create a new rental duration
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when managers decide to create a new rental category.	
2. A manager specifies the name, the minimum and maximum duration and the shorter rental duration.	
	3. Verifies that the rental duration doesn't already exist and the previous rental category exists.
	4. Creates a new rental duration in the specified place
5. A manager specifies the prices for each applicable car group and the newly created rental duration	
	6. Checks that price for a car group is lower or equal in longer durations and higher or equal in shorter durations.
	7. Records the price for each car group introduced.
	8. Each EU-rent branch is notified of the new car group duration prices. SUCCESS EXIT

Extensions:
3a. The rental duration already exists: FAILURE EXIT
3a. The previous rental duration doesn't exist: FAILURE EXIT

Use case: Create a car group duration price
Actors: Manager (initiator)
Overview: Create a car group duration price
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when managers decide to establish a price for a car group and a duration.	
2. A manager specifies the car group, the duration and the price	
	3. Verifies that the car group duration price doesn't already exist.
	4. Create a new car group duration price.
	5. Each EU-rent branch is notified of the new car group duration price. SUCCESS EXIT

Extensions:
3a. The car group duration price already exists: FAILURE EXIT
3b. The car group doesn't exist: FAILUE EXIT
3c. The duration doesn't exist: FAILURE EXIT

Use case: Change price for a car group duration price
Actors: Manager (initiator)
Overview: Change price for a car group duration price
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when managers decide to change the price for the pair car group and duration price.	
2. A manager specifies the car group, the duration and the new price	
	3. Verifies that the car group duration price exist.
	4. Change the car group duration price.
	5. Each EU-rent branch is notified of the change of the duration price. SUCCESS EXIT

Extensions:
3a. The car group duration price doesn't exist: FAILURE EXIT

Car allocation

Use case: Allocate cars to reservations
Actors: -
Overview: Assign cars to next-day reservations
Type: Primary and essential, temporal

Typical course of events:

Actor action	System responsibility
1. At the end of the day, it is time to allocate cars to rental requests due for pick-up the following working day.	
	2. Cancels all reservations not picked-up during the day.
	3. For all available cars, checks if the car must be sold (More than one year old or 40,000 km). *a.* If it must, **Initiate** *Sell a car*
	4. Calculates for each car group: o Availability, availability per model o Number of demands, number of demands per model. o cars available for upgrade
	5. Gets next-day reservations of members of the loyalty incentive scheme. Between them order by guaranteed rental and lastly by time of reservation.
	6. For each customer from 4, **Initiate** *allocate a car for a reservation*
	7. Get unresolved next-day reservations of guaranteed rentals. Between them order by time of reservation.
	8. For each customer from 6, **Initiate** *allocate a car for a reservation*
	9. Get all unresolved next-day reservations. Between them order by time of reservation.
	10. For each customer from 7, **Initiate** *allocate a car for a reservation*
	11. Notify all the branches with which there is an agreement of the end of the assignments.
	SUCCESS EXIT

Use case: Allocate a car for a reservation
Actors: -
Overview: Allocate a car for a next-day reservation
Type: Abstract

Typical course of events:

Actor action	System responsibility
	1. Applies the first action of the following whose condition is true: • Allocate a free upgrade, if availability in the car group < car demand in the car group, and there are remaining upgrades. • Allocate a car of the desired model, if it was specified in the reservation and there are cars available. • Allocate a car of the specified group, belonging to the model with the lowest demand • **Initiate** *Allocate a car with an exception rule* 2. Verify that the end date of the rental is before any scheduled booking of the assigned car for maintenance or transfer. 3. Decrement availability and availability per model of the model of the car group allocated. 4. Decrement number of demands and number of demands per model of the model desired. SUCCESS EXIT

Extensions:
1a. No condition is satisfied: **Initiate** *Allocate a car with an in-extremis rule*
2a. There is some booking: Go to 1 and try with another car assignment

Use case: Allocate a car with an exception rule
Actors: Clerk
Overview: Allocate a car for a next-day reservation
Type: Abstract
Typical course of events:

Actor action	System responsibility
	1. Calculates applicable exception rules.
2. A clerk selects one option.	
	3. Selects one car that meets the requirements of the option selected. 4. Perform the specified action rule. SUCCESS EXIT

Exception rules:

RULE	APPLICATION CONDITION	ACTION
Allocate a car from the capacity reserved for walk-ins	The immediate higher car group has capacity for walk-ins.	The selected car is assigned to the customer reservation. If there is a loyalty member reservation which doesn't have a free upgrade, assignments are exchanged.

Make a bumped upgrade	The second higher car group has surplus	An assigned car of the immediate higher group *x* (from a loyalty member if there is one, any otherwise) is reassigned to the customer reservation. The selected car is assigned to the orphan reservation.
Make a downgrade	The immediate lower car group has surplus.	The selected car is assigned to the customer reservation. The selected car is assigned to the customer reservation.
Allocate a car from another branch	Exists a branch with a relation which: · has an available car of the car group desired · there is enough time for doing the transfer The candidate branch will be the nearest.	Order the car to be transferred (notifying the transferring branch) Assign the car to the customer reservation.
Use a car scheduled for service	Exists a car of the car group desired which have to be serviced and the rental won't take the mileage more than 10% over the normal mileage for service.	Cancel service. Assign the car to the customer reservation.

Use case: Allocate a car with an in-extremis rule
Actors: Branch manager
Overview: Allocate a car for a next-day reservation
Type: Abstract
Typical course of events:

Actor action	System responsibility
	1. Calculates "least bad" in-extremis rules and shows their description.
2. A branch manager selects one option.	
	3. Perform the specified action rule. SUCCESS EXIT

In-extremis rules:

RULE	CHARACTERISTICS	ACTION
Pick-up delay	The pick-up should be delayed until a car is returned and prepared. It is calculated the expected time.	The rental is marked to be pendant of assignment
Rent a car from a competitor	Calculate cost from renting the car from a competitor. Select the cheapest one.	Assign the car to the rental.

Use case: Transfer cars
Actors: Clerk, Head mechanic, Trailer driver
Overview: Cars to be transferred are transported to their destinations.
Type: Essential, temporal
Typical course of events:

Actor action	System responsibility
	1. Once cars have been assigned to reservations in all branches which the concrete branch has agreements, lists (and prints) all the cars needed to be transferred and their destinations.
	2. For each car, change its status to being transferred
3. A clerk picks up the printout and gives it to the head mechanic, who will prepare the cars to be transferred.	
4. The trailer driver transports the cars to the destinations.	
SUCCESS EXIT	

Use case: Receive cars being transferred
Actors: Clerk, Head mechanic, Trailer driver (initiator)
Overview: Cars being transferred are delivered.
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when the trailer driver arrives to a branch where he/she is transporting cars to.	
2. Notifies the head mechanic and cars are queued for preparation.	
3. The head mechanic notifies a clerk of the arrival of cars.	
4. The clerk introduces the registration number of every car.	
	5. Ownership of cars is transferred to the branch and cars are available.
	6. Cars being rented during the day, are queued to be prepared.
	SUCCESS EXIT

Use case: Establish a transfer agreement between branches
Actors: Receiver branch manager (initiator), Transferor branch manager
Overview: One branch asks another branch if it agrees to transfer them cars.
Type: Primary and essential

Typical course of events:

Actor action	System responsibility
1. The use case begins when a branch manager decides to establish a transfer agreement with another branch.	
2. The branch manager –receiver- contacts with a branch manager –transferor- from the other branch. The branch manager asks the transferor if they agree to transfer cars to them when necessary.	
3. The transferor agrees and introduces the receiver branch.	
	4. Verifies that the transfer agreement with the introduced branch doesn't already exist.
	5. A new transfer agreement is recorded.
	6. If the transferor is not a receiver of the receiver: **section** *Introduce transfer agreement data*
	SUCCESS EXIT

Extensions:
3a. The transferor doesn't agree: FAILURE EXIT
4a. The transferor agreement already exists: FAILURE EXIT

Section: Introduce transfer agreement data

Actor action	System responsibility
1. The branch manager introduces expected time to transfer cars and distance between branches.	
	2. Distance and time is recorded for the transfer agreement

Use case: Change transfer agreement data
Actors: Clerk, Branch Manager (initiator)
Overview: Distance to a receiver branch or/and transfer time is changed.
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a branch manager decides to update data from a transfer agreement	
2. A clerk introduces the receiver branch, the new expected time to transfer cars or/and the distance between branches.	
	3. The new data about the transfer agreement is recorded.
	SUCCESS EXIT

Extensions:
3a. The transfer agreement doesn't exist: FAILURE EXIT

Use case: Cancel a transfer agreement between branches
Actors: Transferor Branch Manager (initiator)
Overview: One transferor branch decides to cancel one of its agreements.
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a branch manager decides to cancel a transfer agreement with another branch.	
2. The branch manager introduces the other branch name.	
	3. Verify that the transfer agreement exists.
	4. The transfer agreement is eliminated for future transfers.
	SUCCESS EXIT

Extensions:
3a. The transfer agreement doesn't exist: FAILURE EXIT

Car preparation and maintenance

Use case: Get cars to be prepared
Actors: Head Mechanic (initiator)
Overview: A mechanic gets cars to be prepared.
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when the head mechanic starts his/her working day.	
2. The head mechanic asks for cars to be prepared during that day	
	3. Returns cars to be prepared (that is, available cars assigned to reservations of that day). Cars are returned in order of pick-up.
4. The head mechanic queues the cars to be prepared. SUCCESS EXIT	

Use case: End of car preparation
Actors: Mechanic (initiator), Clerk, Customer
Overview: A car has been prepared for a rental.
Type: Primary and essential

Typical course of events:

Actor action	System responsibility
1. The use case begins when a mechanic finishes the preparation of a car to be rented.	
2. The mechanic tells a clerk the registration number of the car. The clerk introduces the number.	
	3. The status of the car is updated to available.
4. If there is a customer waiting for the car, he/she is notified. SUCCESS EXIT	

Extensions:
3a. Car doesn't exist: FAILURE EXIT

Use case: End of car checking
Actors: Mechanic (initiator), Clerk
Overview: A car has been checked after a rental
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a mechanic finishes the checking of a car being rented.	
2. If the car has been damaged and it is liable to the renter, **initiate** *Detect damages*	
3. The mechanic tells a clerk the current mileage of the car.	
4. The clerk introduces the registration number of the car and its new mileage.	
	5. Checks if the car needs to be serviced (More than 3 months have passed since the last maintenance or it has accumulated 10,000 kilometres since then). a. If the car needs to be serviced, service is scheduled. Status is changed to maintenance scheduled.
	6. Checks if the car needs to be sold (More than one year old or 40,000 km). *a.* If the car needs so and no repairs or maintenance are needed, **initiate** *Sell a car*
	7. If neither repairs, maintenance or selling, checks if there is a today reservation without car and the freed car can be assigned to it. SUCCESS EXIT

Extensions:
5a. Car doesn't exist: FAILURE EXIT

Use case: Detect damages
Actors: Mechanic (initiator), Clerk
Overview: A car has been checked after a rental and damages has been detected
Type: Abstract
Typical course of events:

Actor action	System responsibility
1. A mechanic tells a clerk the rental data corresponding to the car where damages have been detected.	
2. The clerk introduces the data.	
	3. Records a bad experience for the last renter of the car (and the additional drivers if there are). Credit card company must be notified.
	4. If the customer was member of the Loyalty Incentive Scheme he loses his membership.
	5. Check criteria to be blacklisted a. If they are achieved, **initiate** *Cancel all reservations* of the renter due to blacklisting
	6. The credit card provided by the renter is charged for the damages.
	7. Reparation is scheduled and car status is updated. SUCCESS EXIT

Extensions:
3a. Car doesn't exist: FAILURE EXIT

Use case: Schedule maintenance
Actors: Clerk (initiator)
Overview: A clerk establishes a date and a service depot for car maintenance
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a clerk decides to schedule maintenance for a car.	
2. The clerk introduces de registration number of the car and the beginning date of the maintenance.	
	3. Checks if the car needs maintenance.
	4. Checks if the corresponding service depot has capacity for the beginning date.
	5. Maintenance is scheduled for the indicated car. SUCCESS EXIT

Extensions:
3a. Car doesn't need maintenance: FAILURE EXIT
4a. The service depot doesn't have capacity for the indicated day: FAILURE EXIT

Use case: End of car maintenance
Actors: Mechanic (initiator), Clerk
Overview: A car has been serviced.
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a mechanic finishes the maintenance of a car.	
2. The mechanic tells a clerk the registration number of the car. The clerk introduces the number.	
	3. Updates car mileage from last service to current.
	4. Checks if the car needs to be sold (More than one year old or 40,000 km).
	a. If the car needs so, **initiate** *Sell a car*
	b. Otherwise, the car is available
	SUCCESS EXIT

Extensions:
3a. Car doesn't exist: FAILURE EXIT

Use case: End of car repair
Actors: Mechanic (initiator), Clerk
Overview: A car has been serviced.
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a mechanic finishes repairing a car.	
2. The mechanic tells a clerk the registration number of the car. The clerk introduces the number.	
	3. The end of the repair is recorded.
	4. Checks if the car needs to be sold (More than one year old or 40,000 km).
	a. If the car needs so and no maintenance is booked, **initiate** *Sell a car*
	b. The car is available
	SUCCESS EXIT

Extensions:
3a. Car doesn't exist: FAILURE EXIT

Car pick-up and return

Car pick-up

Use case: Pick-up a car
Actors: Customer (initiator), Clerk
Overview: A customer picks-up a car which was reserved.
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when a customer arrives to pick-up a car.	
2. The customer shows his ID to a clerk. The clerk introduces the number	
	3. Verifies that there is reservation due to pick-up for that customer that day and the car is expected to be prepared around fixed time.
4. The customer waits until the car is prepared.	
5. **Initiate** *Handover* of the car assigned to the rental	
	6. If the car is given substantially late, send an apologetic letter to the customer and pay him/her the cost for the non-served period SUCCESS EXIT

Extensions:
3a. The reservation is not due to pick-up because it doesn't exist, it has been cancelled (either because the customer has arrived late and car has been assigned to another rental or other reasons) or it is for a future day: FAILURE EXIT
3b. The car is expected to not be prepared at fixed time.

Actor action	System responsibility
	3b.1. Notifies that the car won't be prepared. Gives the expected time.
3b.2. The clerk notifies the customer that the car won't be prepared at fixed time.	
3b.3. b. The customer decides to wait. GOTO 4. c. The customer decides to cancel the reservation with no cost to him/her.	
3b.4. The clerk introduces the customer's choice.	
	3b.5. The reservation is cancelled if so decided and an apologetic letter is sent. SUCCESS EXIT

Use case: Handover
Actors: Customer (initiator), Clerk, Additional drivers
Overview: Verifies that the renter and the additional drivers can drive the car assigned to their rental.
Type: Abstract
Typical course of events:

Actor action	System responsibility
1. The clerk verifies that the customer (driver) is in an appropriate condition to drive the car	
2. The customer tells the clerks who are the additional drivers, if there are any.	
3. The clerk verifies that each additional driver is in an appropriate condition to drive the car.	
4. An additional driver tells his ID to the clerk and he/she introduces it	
	5. Checks if the driver is a person who has had contact with EU-Rent. *a.* If it exists, verifies that the driver has not been blacklisted *b.* **Initiate** *Introduce a new EU-rent costumer/driver,* otherwise
6. The additional drivers sign an 'additional drivers authorization'	
7. Repeat 7,8 and 9 while there are additional drivers.	
8. The customer signs the rental contract.	
9. The clerk confirms the pick-up.	
	10. The status of the rental is changed to open. SUCCESS EXIT

Extensions:
1a. The driver is not in an appropriate condition to drive the car (either appears to be under the influence of alcohol or drugs, or is not physically able to drive the car safely): FAILURE EXIT
3a. Any of the additional drivers is not in an appropriate condition to drive the car: FAILURE EXIT
6a. Any of the additional drivers doesn't sign the authorization: FAILURE EXIT
8a. The renter doesn't sign the authorization: FAILURE EXIT

Use case: Free cars
Actors: -
Overview: At the end of each day, cars assigned to reservations which have not been picked up are freed.
Type: Essential, temporal

Typical course of events:

Actor action	System responsibility
1. At the end of each working day, an automatic process is initiated to free all the cars assigned to reservations which have not been picked up.	
	2. Obtains reservations which have not been picked up during that day.
	3. Selects one reservation and change its status to cancelled. The car assigned is released.
	4. If it was a guaranteed rental, one day's rental is charged to the guaranteeing credit card.
	5. Repeat 3 and 4 while there are not cancelled reservations from 2. SUCCESS EXIT

Car return

Use case: Return of a car
Actors: Customer (initiator), Clerk
Overview: A car has been returned to an EU-Rent branch
Type: Primary and essential

Typical course of events:

Actor action	System responsibility
1. The use case begins when a customer returns the car that he/she was renting.	
2. The customer leaves the car to a mechanic and starts the car checking.	
3. The customer tells a clerk his ID and beginning date of the rental.	
	4. The system verifies that the rental exists and stores actual time as return time.
	5. Verifies that the car has been returned to the agreed drop-off branch.
	6. Verifies that the car has been returned on time.
	7. If the drop-off branch is different from the pick-up branch, car ownership is transferred.
	8. Calculates basic cost of the rental (up to the duration and car group): **initiate** *Calculate best price.*
	9. Add to the basic cost extras and corresponding taxes to the country.
	10. Shows total cost and asks for payment type (cash, credit card)
11. The clerk asks the customer for payment type and number of credit card to be charged in case of car damages.	
12. The clerk introduces customer data and the customer pays the rental.	
	13. The rental is closed. SUCCESS EXIT

Extensions:

4a. The rental doesn't exist: FAILURE EXIT

5a. The car has not been returned to the agreed drop-off branch.

Actor action	System responsibility
	5a.1. Adds a drop-off penalty to the total cost of the rental. CONTINUE

6a. The car has been returned substantially before the agreed drop-off time/day.

Actor action	System responsibility
	6a.1. Takes it into account to calculate the rental charge of the actual period of rental. CONTINUE

6b. The car has been returned after the agreed drop-off time/day.

Actor action	System responsibility
	6b.1. The cost is incremented by one of this ways: • An hourly charge up to 6 hours delay • A daily charge upon 6 hours delay 6b.2. A bad experience is recorded with seriousness depending on the delay interval. 6b.3. If the customer was member of the Loyalty Incentive Scheme, he loses his membership. 6b.4. Check criteria to be blacklisted a. If they are achieved, **initiate** *Cancel all reservations* of the customer due to blacklisting CONTINUE

Use case: Control late returns
Actors: Clerk, Customer,
Overview: At the end of each day, status of late returns is checked.
Type: Essential, temporal
Typical course of events:

Actor action	System responsibility
1. At the end of each working day, an automatic process is initiated to control the state of cars which should have been returned and are not.	
	2. Obtain rentals which should have been returned during that day in that branch and have not been returned to that or any other. 3. Select one rental and show the details (and also customer details).
4. A clerk contacts with the customer.	
	5. Repeat 3 and 4 while there are rentals from 2 whose renter has not been contacted. 6. Obtain rentals which should have been returned during 3 days before in that branch and have not been returned to that or any other. 7. Show their data.
8. The clerk contacts the police and tells them the data of the disappeared renters. SUCCESS EXIT	

Extensions:
4a. The customer can't be contacted: CONTINUE with the other rentals

Car management

Use case: Buy a car
Actors: Clerk (initiator)
Overview: A car is bought
Type: Abstract
Typical course of events:

Actor action	System responsibility
1. The use case begins when a car need to be purchased.	
2. The clerk specifies the model required to buy.	
	3. Order a car of this model to the supplier. Save order details. SUCCESS EXIT

Extensions:
3a. The model is not in the authorization list: FAILURE EXIT

Use case: Receive a car
Actors: Clerk (initiator)
Overview: An ordered car arrives
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when an ordered car arrives to a branch.	
2. The clerk specifies order ID and car data (registration number)	
	3. Verifies that exists a pendant order and the information is consistent. Close the order.
	4. A new car is recorded with the information supplied. The car is available and the branch has its ownership. SUCCESS EXIT

Extensions:
3a. The order ID doesn't exist: FAILURE EXIT
3b. The information is not consistent (existing registration number): FAILURE EXIT

Use case: Sell a car
Actors: -
Overview: A car is sold.
Type: Abstract

Typical course of events:

Actor action	System responsibility
1. The use case begins when a car need to be sold.	
	2. Verifies that branch has car ownership and the car is not assigned to any current rental.
	3. The car is available to be sold. SUCCESS EXIT

Extensions:
3a. The car doesn't exist: FAILURE EXIT
3b. Branch doesn't have car ownership: FAILURE EXIT
3c. Car is assigned to a current rental: FAILURE EXIT

Use case: Control number of cars
Actors: Branch manager (initiator)
Overview: From time to time, number of cars in each branch is controlled: cars are bought, sold or transferred if necessary.
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when the branch manager decides to control the number of cars of the branch.	
2. The branch manager asks for car number control.	
	3. Calculates number of cars which are owned by the branch (taking also into account ordered cars)
	4. For each car group,
	a. If there is a surplus over 10%, **Initiate** *Reduce number of cars*
	b. If there is a lack over 10%, **Initiate** *Increase number of cars*
	SUCCESS EXIT

Use case: Increase number of cars
Actors: Branch manager
Overview: Number of branch cars of a certain car group is increased by transferring cars from other branches or buying new ones.
Type: Abstract

Typical course of events:

Actor action	System responsibility
	1. Obtain availability of branches with a transfer agreement by distance/time priority.
	2. Select the first one with surplus of this car group.
	3. Order exceeding cars while there is provider surplus and the receiver branch is below 10% of its quota. Notify the transferring branch.
	4. Repeat 2 and 3 while the branch is still below 10% of its quota and there are branches with a transfer agreement with surplus of the car group.
	5. Check if the car group quota is below 10%: a. If it is, asks for a model to buy. b. SUCCESS EXIT
6. The branch manager selects a model of the car group	
	7. **Initiate** *Buy a car* of the model
	8. GOTO 5.

Use case: Reduce number of cars
Actors: -
Overview: Number of branch cars of a certain car group is decreased by transferring cars to other branches or selling cars.
Type: Abstract
Typical course of events:

Actor action	System responsibility
	1. Obtain availability of branches with a transfer agreement by distance/time priority.
	2. Select the first one with lack of this car group.
	3. Mark cars as pending of transfer while the provider branch is over 10% of its quota (and there are available cars) of that car group and the receiver has a lack of this car group.
	4. Repeat 2 and 3 while the branch is still over 10% of its quota (and there are available cars) and there are branches with a transfer agreement with lack of the car group.
	5. While the car group quota is over 10%, **initiate** *Sell a car* with the oldest car in the car group. SUCCESS EXIT

Use case: Confirm car sale
Actors: Manager (initiator)
Overview: The branch manager approves the sale of a car to be sold
Type: Primary, essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when managers decide to sell a car.	
2. The manager specifies the car he wants to sell.	
	3. Checks that the car belongs to the branch and that it is pending to be sold
	4. The state of the car changes to sold, with current date as disposal date.

Extensions:
3a. The car does not belong to the branch: FAILURE EXIT
3b. The car is not to be sold: FAILURE EXIT

Branch management

Use case: Create a branch
Actors: Manager (initiator)
Overview: EU-Rent decides to open a new branch
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when managers decide to establish a new branch.	
2. A manager specifies the location of the new branch.	
	3. Checks if information about the country exists. a. If it doesn't, **initiate** *Record country information.*
4. The manager introduces the characteristics of the new branch (name, type, country, quota per car group)	
	5. Verifies that the branch doesn't already exist.
	6. Create a new branch with the information provided.
	SUCCESS EXIT

Extensions:
3a. The branch already exists: FAILURE EXIT

Use case: Record country information
Actors: Manager (initiator)
Overview: Record information of a new country with EU-Rent branches.
Type: Abstract

Typical course of events:

Actor action	System responsibility
1. A manager specifies the characteristics of the country of the new branch (name, requirements for mechanical condition and emissions)	
	2. Verifies that the country doesn't already exist.
	3. Create a new country with the information provided.
	SUCCESS EXIT

Extensions:
2a. The country already exists: FAILURE EXIT

Car group and models management

Use case: Create a new car group
Actors: Manager (initiator)
Overview: Record information of a new car group.
Type: Primary and essential.
Typical course of events:

Actor action	System responsibility
1. The use case begins when managers decide to create a new car group	
2. A manager specifies characteristics of the new car group (name, previous –immediately worse- group if there is one)	
	3. Verifies that preceding car group exists, unless it is not provided (then, it's the first).
	4. Verifies that the new car group doesn't already exist.
	5. Create the car group and insert it in the appropriate place
6. Introduce one car model for the group.	
	7. Check if the car model exists,
	a. If it exists, changes the car model of group
	b. Initiate *Add a car model* and establish as its car group the newly created, otherwise.
8. Repeat 5 and 6 while more models are wanted to be assigned.	
	9. For each branch, establish a (default) minimum quota of the new car group.
	SUCCESS EXIT

Extensions:
3a. The preceding car group doesn't exist: FAILURE EXIT
4a. The car group already exists: FAILURE EXIT

Use case: Add a car model
Actors: Manager (initiator)
Overview: Record information of a new car model.
Type: Abstract
Typical course of events:

Actor action	System responsibility
1. The use case begins when it is decided to create a new car model	
2. A manager specifies characteristics of the new car model (name, technical characteristics)	
	3. Verifies that the new car model doesn't already exist.
	4. Create the car model with the supplied information SUCCESS EXIT

Extensions:
3a. The car model already exists: FAILURE EXIT

Use case: Change a car model group
Actors: Manager (initiator)
Overview: Change car model group.
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when it is decided to change a model of car group	
2. A manager specifies the model and the new car group.	
	3. Change the car model of group. SUCCESS EXIT

Extensions:
3a. The car model doesn't exist: FAILURE EXIT
3b. The car group doesn't exist: FAILURE EXIT

Use case: Change car group quota
Actors: Branch manager (initiator)
Overview: Establish quota for a car group in an EU-rent branch
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when the branch manager decides to change a car group quota.	
2. The manager specifies the car group and the new quota.	
	3. Change the quota for the car group. SUCCESS EXIT

Extensions:
3a. The car group doesn't exist: FAILURE EXIT

Performance indicators management

Use case: Add a performance indicator
Actors: Manager (initiator)
Overview: Add a performance indicator for a country and type of branch
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when managers decide to introduce a new performance indicator for a country and type of branch.	
2. A manager introduces the country name and type of branch.	
	3. Verifies that the type of branch exists in the country.
4. The manager introduces the indicator's name.	
	5. Checks if the indicator exists. a. If it doesn't exist, **section** *Describe indicator*
6. The manager introduces target value.	
	7. Verifies that the indicator hasn't been already defined for this country and type of branch 8. Record the information for the new target value of the indicator. SUCCESS EXIT

Extensions:
3a. The type of branch doesn't exist in the country (or the country or type of branch don't exist): FAILURE EXIT
7a. The indicator has already been defined for this combination: FAILURE EXIT

Section: Describe indicator

Actor action	System responsibility
1. The manager introduces a description for the indicator.	
	2. Records new indicator information.

Use case: Change a target value for a performance indicator
Actors: Manager (initiator)
Overview: Change a target value for a performance indicator for a country and type of branch
Type: Primary and essential

Typical course of events:

Actor action	System responsibility
1. The use case begins when managers decide to change target value for a performance indicator for a country and type of branch.	
2. A manager introduces the country name and type of branch.	
	3. Verifies that the type of branch exists in the country.
4. The manager introduces the name for the indicator and its new target value.	
	5. Verifies that the indicator already exists.
	6. Modify the target value for the indicator.
	SUCCESS EXIT

Extensions:
3a. The type of branch doesn't exist in the country (or the country or type of branch doesn't exist): FAILURE EXIT
5a. The indicator doesn't exist: FAILURE EXIT

Use case: List information of a performance indicator
Actors: Branch Manager (initiator)
Overview: List information of a concrete performance indicator in a branch
Type: Primary and essential
Typical course of events:

Actor action	System responsibility
1. The use case begins when the branch manager wants to get information of a certain performance indicator.	
2. The manager introduces the name of the indicator	
	3. Verifies that the indicator exists.
	4. List information of the concrete indicator.
	SUCCESS EXIT

Extensions:
3a. The indicator doesn't exist: FAILURE EXIT

Use case: Get alert indicators and execute actions
Actors: Branch Manager (initiator)
Overview: List indicator which doesn't meet the target and apply actions trying to solve the problems
Type: Primary and essential

Typical course of events:

Actor action	System responsibility
1. The use case begins when the branch manager wants to get indicators which don't meet the target.	
	2. List information of indicators which doesn't meet the target.
3. The branch manager decides to apply a certain *action* to solve a problem.	
4. *Executes the action*	
5. Repeat 3 and 4 while there are problems to be solved.	
SUCCESS	

Note: *Executes the action* is a quite general term and could be: sell a car, buy a car, change quota for a car group…

5. STATIC MODEL

Overview

The central part of the static model is the class diagram where concepts relevant to the system and their relationships are represented.

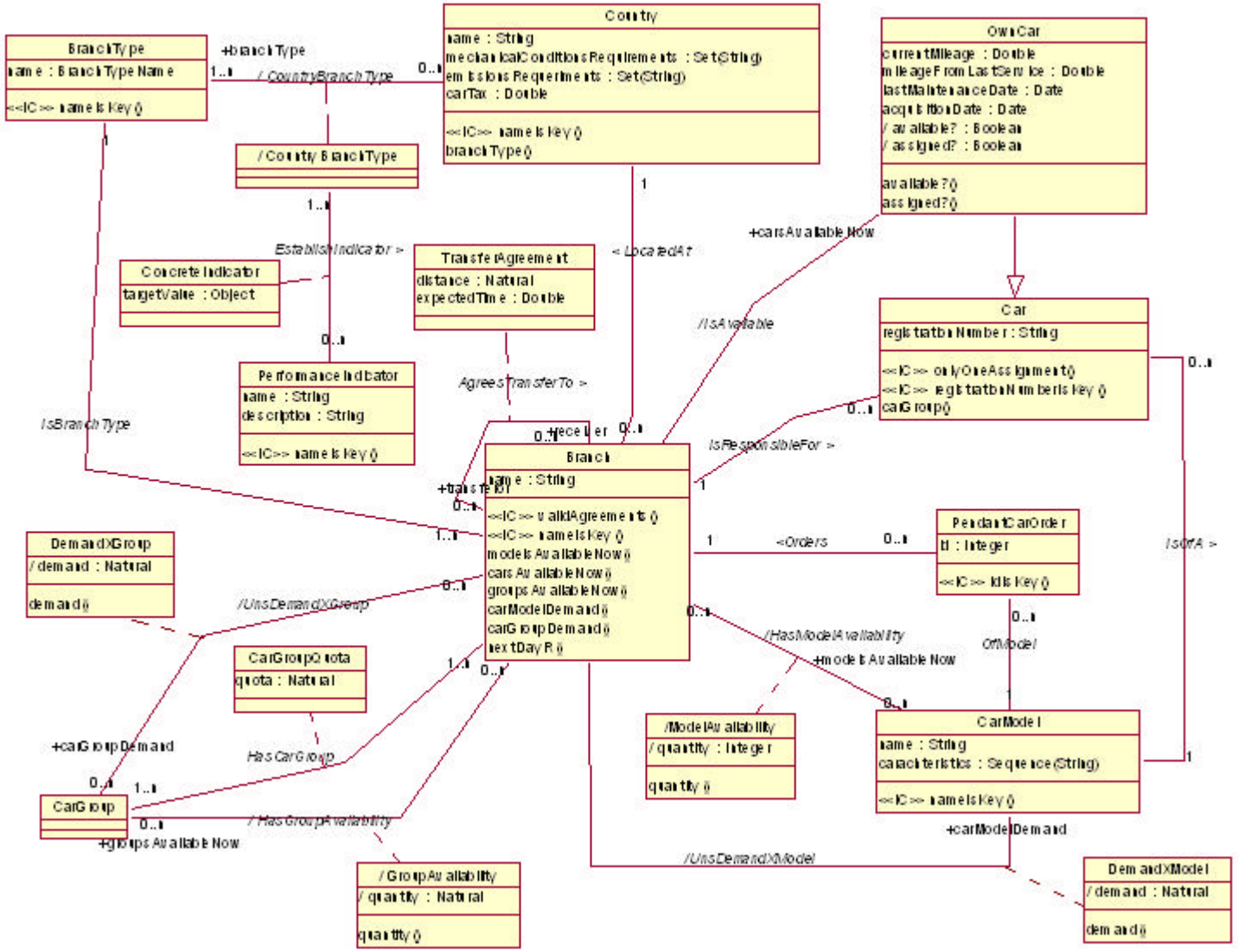
On the other hand, a complete conceptual schema must include the definition of all relevant integrity constraints. One simple way to do so is by assigning an operation with the stereotype <<IC>> to each constraint, as suggested in [IC-Ol03]. This technique is the one applied in this project together with an analogous mechanism for the definition of derived elements as described in [DR-Ol03].

Diagrams

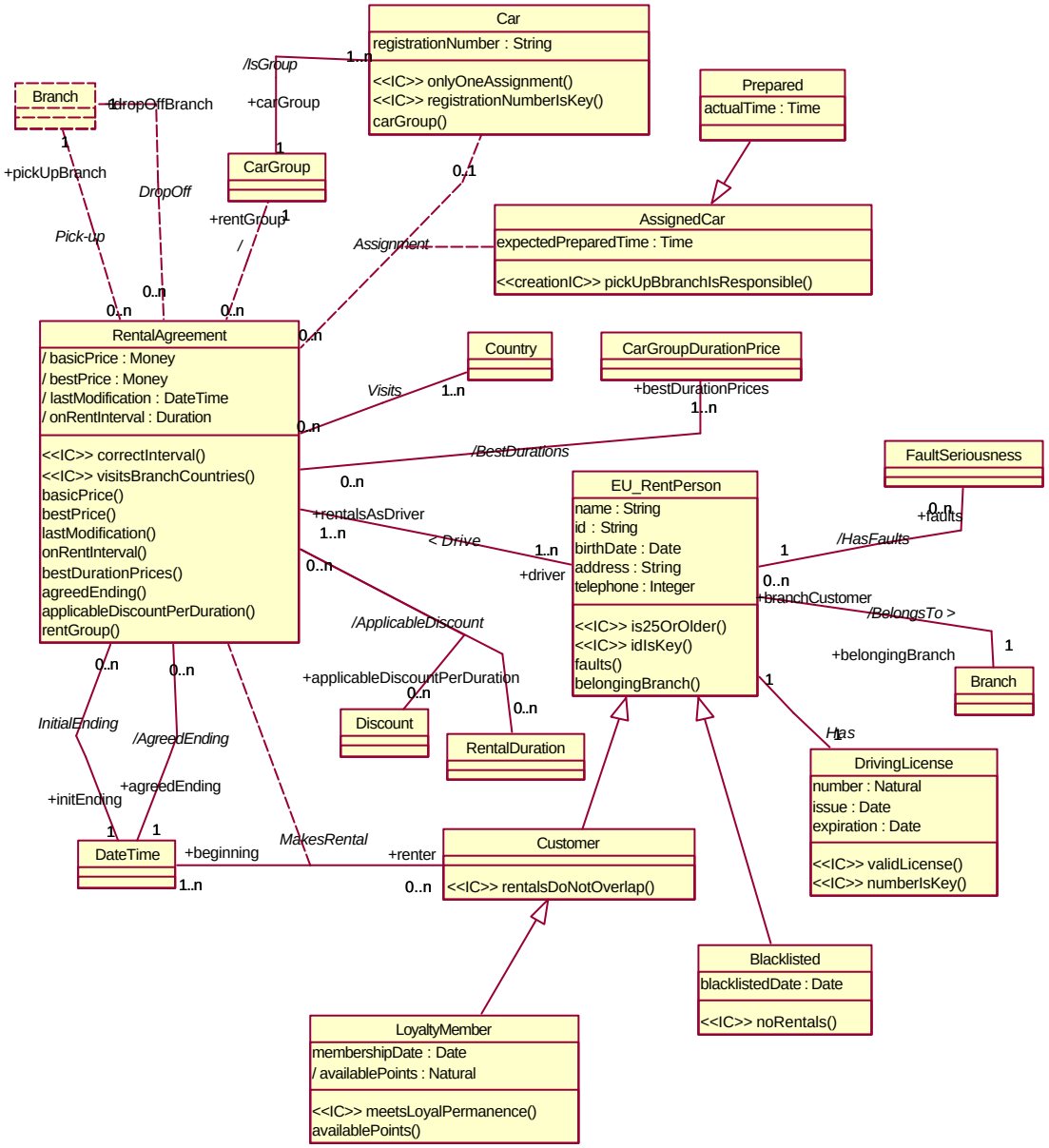
The class diagram is divided in thematic areas to ease understanding. These are the following:

- Branch
- Rental Agreement
- Rental Agreement subclasses
- Cars, Discounts and Enumerations

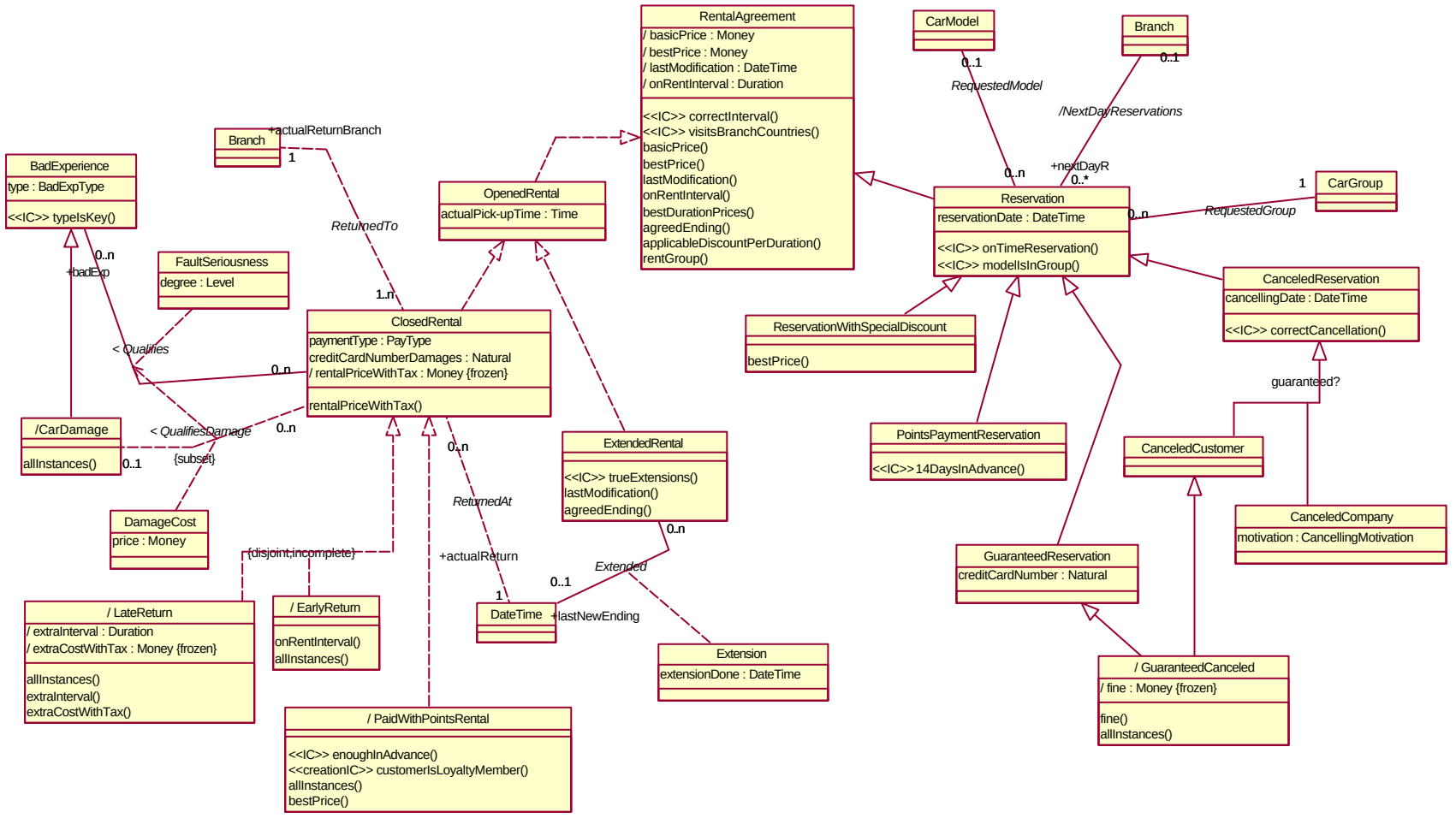
Branch


BranchType
name : BranchType Name
<<IC>> nameIsKey()
+branchType
1..1
/CountryBranchType
0..1
Country
name : String
mechanicalConditionsRequirements : Set(String)
emissionsRequirements : Set(String)
carTax : Double
<<IC>> nameIsKey()
branchType()
/CountryBranchType
1..1
1
OwnCar
currentMileage : Double
mileageFromLastService : Double
lastMaintenanceDate : Date
acquisitionDate : Date
/ available? : Boolean
/ assigned? : Boolean
available?()
assigned?()
+carsAvailableNow
EstablishIndicator >
ConcreteIndicator
targetValue : Object
TransfetAgreement
distance : Natural
expectedTime : Double
< LocatedAt
/IsAvailable
Car
registrationNumber : String
<<IC>> onlyOneAssignment()
<<IC>> registrationNumberIsKey()
carGroup()
0..1
0..1
PerformanceIndicator
name : String
description : String
<<IC>> nameIsKey()
AgreesTransferTo >
IsBranchType
0..1
+receiver
0..1
IsResponsibleFor >
Branch
name : String
<<IC>> validAgreements()
<<IC>> nameIsKey()
modelsAvailableNow()
carsAvailableNow()
groupsAvailableNow()
carModelDemand()
carGroupDemand()
nextDayR()
+transferor
0..1
1..1
1
<Orders
0..1
PendantCarOrder
tl : Integer
<<IC>> idIsKey()
IsOfA >
DemandXGroup
/ demand : Natural
demand()
/UnsDemandXGroup
0..1
0..1
0..1
/HasModelAvailability
+modelsAvailableNow
OfModel
0..1
1
CarGroupQuota
quota : Natural
1..1
0..1
/ModelAvailability
/ quantity : Integer
quantity()
CarModel
name : String
characteristics : Sequence(String)
<<IC>> nameIsKey()
1
+carGroupDemand
0..1
1..1
HasCarGroup
CarGroup
0..1
/HasGroupAvailability
+groupsAvailableNow
+carModelDemand
/UnsDemandXModel
/GroupAvailability
/ quantity : Natural
quantity()
DemandXModel
/ demand : Natural
demand()

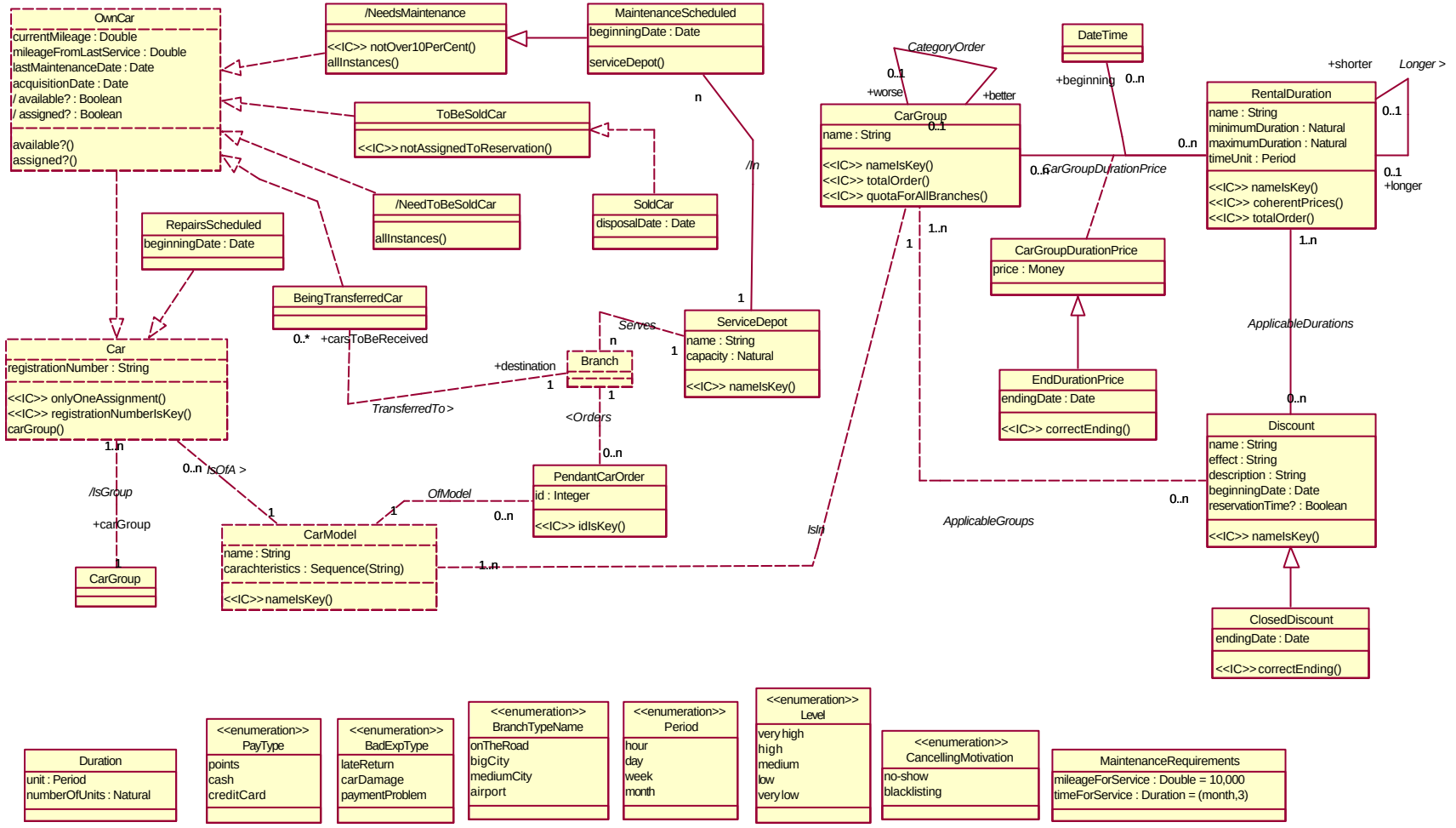
Rental Agreement



Rental Agreement subclasses



Cars, Discounts and Enumerations



Complete specification of operations associated to derived elements and integrity constraints

Branch

```
context   Branch:: validAgreements() :   Boolean
post:
          result=self.receiver->excludes(self) and self.transferor->
            excludes(self) and self.receiver->forAll(b|b.receiver->
            includes(self) implies
           let tA1:TransferAgreement=TransferAgreement.allInstances()->
            select(tA| (tA.receiver=self and tA.transferor=b))
           let tA2:TransferAgreement=TransferAgreement.allInstances() ->
            select(tA| tA.transferor=self and tA.receiver=b)
          in
           tA1.distance(km)=tA2.distance(km) and tA1.expectedTime(h)=
            tA2.expectedTime(h)


context   Branch:: nameIsKey() :   Boolean
post:
          result=Branch.allInstances()->
            select(b|b.name=self.name)->size()=1


context   Branch:: modelsAvailableNow() :      Set(CarModel)
post:
          result= self.carsAvailableNow.carModel->asSet()


context   Branch:: carsAvailableNow(car : Car) :     Set(Car)
post:
          result=self.car->select(oclIsKindOf(OwnCar))->
            select(c:Car|  c.oclAsType(OwnCar).available?)


context   Branch:: groupsAvailableNow() :      Set(CarGroup)
post:
          result=self.modelsAvailableNow.carGroup->asSet()


context   Branch:: carModelDemand() :    Set(CarModel)
post:
          result=carModel.allInstances()


context   Branch:: carGroupDemand() :    Set(CarGroup)
post:
          result=CarGroup.allInstances()


context   Branch:: nextDayR() :    Set(Reservation)
post:
          Reservation.allInstances()->select(r|r.beginning.date()=
          tomorrow())->select(r|r.pickUpBranch=self)
```

BranchType

```
context   BranchType:: nameIsKey() :     Boolean
```

```
post:
          result=BranchType.allInstances()->
            select(b|b.name=self.name)->size()=1
```

Country

```
context   Country:: nameIsKey() :  Boolean
post:
          result=Country.allInstances->
            select(b|b.name=self.name)->size()=1


context   Country:: branchType() : Set(BranchType)
post:
          result=self.branch.branchType->asSet()
```

PerformanceIndicator

```
context   PerformanceIndicator:: nameIsKey() : Boolean
post:
          result=PerformanceIndicator.allInstances()->
            select(b|b.name=self.name)->size()=1
```

ModelAvailability

```
context  ModelAvailability:: quantity() :      Natural
post:
          result=self.branch.carsAvailableNow->
            select(carModel=self.carModel)->size()
```

GroupAvailability

```
context   GroupAvailability:: quantity() :     Integer
post:
          return= self.branch.modelAvailability->
            select(mA |mA.carModel.carGroup= self.carGroup)->sum(quantity)
```

DemandXModel

```
context   DemandXModel:: demand() :      Integer
post:
          let pendantRes:Reservation= Reservation.allInstances()->
          select(r|r.beginning.date()=tomorrow())->select(r|
            r.pickUpBranch=self.branch and r.car->isEmpty())
          in
          result=pendantRes.requestedModel->select(m|m=d.carModel)->size()
```

DemandXGroup

```
context   DemandXGroup:: demand() :      Integer
post:
          let pendantRes:Reservation= Reservation.allInstances()->
          select(r|r.beginning.date()=tomorrow())->select(r|
            r.pickUpBranch=self.branch and r.car->isEmpty())
          in
          result=pendantRes.requestedGroup->select(m|m=d.carGroup)->size()
```

RentalAgreement

```
context   RentalAgreement:: correctInterval() :      Boolean
post:
          result=self.beginning< self.initEnding and self.actualReturn>
            self.beginning
documentation:
          Ending date-time of a rental (actual and expected) must be
          after the beginning dates-times (actual and expected) of the
          rental


context   RentalAgreement:: visitsBranchCountries() :      Boolean
post:
          result = self.Countries->includes(self.PickUpBranch.Country) and
            self.Countries->includes(self.DropOffBranch.Country)


context   RentalAgreement:: basicPrice() :     Money
post:
          result= self.bestDurationPrices-> iterate(elem;
            tup : Tuple {accInterval: Duration=self.onRentInterval,
            accPrice:     Money=0} |
            let timeMax:Duration= durationT(elem.timeUnit,
            elem.maximumDuration)
            let timeMin:Duration= durationT(elem.timeUnit,
            elem.minimumDuration)
            let numInt:Integer =
              if tup.accInterval >= timeMax then
                tup.accInterval/timeMax
              else
                tup.accInterval/timeMin
            in
               Tuple {accInterval:Duration=
               (if tup.accInterval >= timeMax then
                      tup.accInterval%timeMax
                 else
                      tup.accInterval%timeMin
                 endif),
                  accPrice:Money= tup.accPrice+numInt*elem.price}
          ).accPrice
documentation:
          Best price for the rental duration since the last modification
            (rental duration was changed) without discounts
```

```
context   RentalAgreement:: bestPrice() :      Money
post:
          let bestRentalDiscountPerDuration(rd:RentalDuration, basicPrice:
             Money): Set(Discount)= self.applicableDiscountPerDuration(rd)
            ->reject(disAct: Discount|
            self.applicableDiscountPerDuration(rd) ->
            exists(disOther:Discount| apply(disOther, basicPrice) <
            apply(disAct, basicPrice))
          in
          result= self.bestDurationPrices->iterate(elem;
            tup : Tuple {accInterval: Duration=self.onRentInterval,
            accPrice:   Money=0} |
            let timeMax:Duration=     durationT(elem.timeUnit,
            elem.maximumDuration)
            let timeMin:Duration=     durationT(elem.timeUnit,
            elem.minimumDuration)
            let numInt:Integer =
              if tup.accInterval >= timeMax then
                 tup.accInterval/timeMax
              else
                 tup.accInterval/timeMin
            in
            Tuple {accInterval:Duration=
            (if tup.accInterval >= timeMax then
                 tup.accInterval%timeMax
               else
                 tup.accInterval%timeMin
               endif),
              accPrice:Money= tup.accPrice+
               numInt*self.bestRentalDiscountPerDuration
              (elem.rentalDuration, elem.price)->any()}).accPrice


context   RentalAgreement:: lastModification() :     DateTime
post:
          result =
          if self.oclIsTypeOf(Resevation) then
            self.reservationDate
          else
           self.beginning
          end if


context   RentalAgreement:: onRentInterval() : Duration
post:
          result= self.agreedEnding-self.beginning
```

```
context   RentalAgreement:: bestDurationPrices() :
      Set(CarGroupDurationPrice)
post:
          let applicableDuration: Set(CarGroupDurationPrice)=
            self.rentGroup.carGroupDurationPrice->
            select(cg: CarGroupDurationPrice | cg.beginning<= self.ending
            and (cg.oclIsTypeOf(EndDurationPrice) implies
            cg.oclAsType(EndDurationPrice).endingDate >=
            self.lastModification)
          let bestCurrentDuration: Set(CarGroupDurationPrice)=
            applicableDuration->reject(cgCur: CarGroupDurationPrice|
            applicableDuration-> exists(cgOther:CarGroupDurationPrice|
            cgOther.rentalDuration=cgCur.rentalDuration and
            cgOther.carGroup= cgCur.carGroup and
            cgOther.price<cgCur.price))
          in
          result= self.bestCurrentDuration->
                  sortedBy(rentalDuration.shorter)


context   RentalAgreement:: agreedEnding() :   DateTime
post:
          result= initEnding


context   RentalAgreement:: applicableDiscountPerDuration(rd :
RentalDuration) : Set(Discount)
post:
          let rentalApplicableDiscount: Set(Discount)=
             self.rentGroup.discount->select(dis: Discount |
             dis.beginningDate<= self.ending and
             (dis.oclIsTypeOf(ClosedDiscount) implies
             dis.oclAsType(ClosedDiscount).endingDate >=
             self.lastModification and applicable(dis, self.renter)
          in
          result=  self.rentalApplicableDiscount->
            select(dis:Discount| dis.rentalDuration=rd)
documentation:
          Defining operation of the association applicable
          Discount


context   RentalAgreement:: rentGroup() :      carGroup
post:
          if self.oclIsKindOf(Reservation) then
            if self.car->isEmpty() or self.car.carGroup<>
                   self.carGroup.worse then
             result=self.carGroup
            else
              result=self.carGroup.worse
            end if
          else
             result=self.car.carGroup
          end if
```

Reservation

```
context   Reservation:: onTimeReservation() :  Boolean
post:
          Result=self.reservationDate < self.beginning
documentation:
          Reservation date of a rental must be previous to the
          beginning date.


context   Reservation:: modelIsInGroup() :     Boolean
post:
          result=self.carModel->notEmpty() implies
            self.carModel.carGroup=self.carGroup
```

PointsPaymentReservation

```
context   PointsPaymentReservation:: 4DaysInAdvance() :    Boolean
post:
          result=self.beginning-self.reservationDate>=day(14)
```

ReservationWithSpecialDiscount

```
context   ReservationWithSpecialDiscount:: bestPrice() :   Money
post:
          let reservationTimeDiscountPerDuration(rd: RentalDuration)
          =self.applicableDiscountPerDuration->select(d|d.reservationTime)
          let bestRentalDiscountPerDuration(rd:RentalDuration,
           basicPrice: Money): Discount=
            self.rentalApplicableDiscountPerDuration(rd)->
            reject(disAct: Discount|
            self.rentalApplicableDiscountPerDuration(rd)->
            exists(disOther:Discount|
            apply(disOther, rd).isBetter( apply(disAct, rd)))->any()
          in
          result= self.bestDurationPrices->iterate(elem;
            tup : Tuple {accInterval: Duration=self.onRentInterval,
            accPrice:   Money=0} |
            let timeMax:Duration=     durationT(elem.timeUnit,
             elem.maximumDuration)
            let timeMin:Duration=     durationT(elem.timeUnit,
             elem.minimumDuration)
            let numInt:Integer =
              if tup.accInterval >= timeMax then
                 tup.accInterval/timeMax
              else
                 tup.accInterval/timeMin
            in
            Tuple {accInterval:Duration=
            (if tup.accInterval >= timeMax then
                  tup.accInterval%timeMax
             else
                  tup.accInterval%timeMin
             endif),
              accPrice:Money= tup.accPrice+numInt*
            apply(self.bestRentalDiscountPerDuration(elem.rentalDuration,
            elem.price), elem.rentalDuration)}).accPrice
```

CanceledReservation

```
context   CanceledReservation:: correctCancellation() :    Boolean
post:
          result=self.cancellingDate>=self.reservationDate and
          self.cancellingDate <=self.beginning
documentation:
          Cancelling date of a reservation must be after or equal to the
          reservation date and before the beginning date or equal to it.
```

GuaranteedCanceled

```
context    GuaranteedCanceled:: fine() : Money
post:
          if self.beginning=self.cancellingDate then
              result= self.bestDurationPrices->select(cGDP |
          not(cGDP.oclIsTypeOf(EndDurationPrice)) and
            cGDP.RentalDuration.timeUnit= Period::day and
            cGDP.RentalDuration.minimumDuration=1)->first().price
          else
            result=0
          end if
documentation:
          A fine of one day rental must be paid if the rental
          was guaranteed and the cancelling date is the same day
          as the expected beginning of the rental.
          Otherwise, no fine must be paid.


context   GuaranteedCanceled:: allInstances() :
      Set(GuaranteedCanceled)
post:
          result=CanceledCustomer.allInstances()->
            intersection(GuaranteedReservation.allInstances())
```

ExtendedRental

```
context   ExtendedRental:: trueExtensions() :  Boolean
post:
          let extensions:Extension=self.newEndings
          in
            result= extensions.extension->sortedBy(e|
            e.extensionDone.dateTime).newEndings= extensions
            and extensions->forAll(ext| ext.extensionDone > self.beginning)
            and self.newEndings->forAll(d|d>self.initEnding)


context   ExtendedRental:: lastModification() :      DateTime
post:
          result=self.Extension->sortedBy(extensionDone)->last()


context   ExtendedRental:: agreedEnding() :    DateTime
post:
          result=self.newEndings->last()
```

ClosedRental

```
context   ClosedRental:: rentalPriceWithTax() :      Money
post:
          result= self.bestPrice * self.actualReturnBranch.country.carTax
```

PaidWithPointsRental

```
context   PaidWithPointsRental:: enoughInAdvance() : Boolean
post:
          result= self.oclIsTypeOf(Reservation) and (self.beginning.day()-
            self.oclAsType(Reservation).reservationDate.day())>=day(14)


context   PaidWithPointsRental:: customerIsLoyaltyMember() :     Boolean
post:
          result= self.renter.oclIsType(LoyaltyMember)


context   PaidWithPointsRental:: allInstances():
      Set(PaidWithPointsRental)
post:
          result= ClosedRental.allInstances->select(cR|cR.paymentType=
             payType::points)


context   PaidWithPointsRental:: bestPrice() : Money
post:
          result=basicPrice
```

LateReturn

```
context   LateReturn:: allInstances() :  Set(LateReturn)
post:
          result= ClosedRental.allInstances()->select(cR|
            cR.actualReturn > cR.agreedEnding)


context   LateReturn:: extraCostWithTax() :    Money
post:
          let timeUnit: Period=
            if self.extraInterval.unit=Period::hour and
          self.extraInterval.numberOfUnits <= 6 then
              Period::hour
            else
             Period::day
            endif
          in
          let durationPrice: Money= self.bestDurationPrices->
            select(cGDP | not(cGDP.oclIsTypeOf(EndDurationPrice)) and
            cGDP.timeUnit= timeUnit and  minimumDuration=1)->first().price
          let extraPrice: Money= durationPrice*(extraInterval/
            durationT(timeUnit,1))+ durationPrice*(extraInterval%
            durationT(timeUnit,1))
          in
          result=  extraPrice * self.actualReturnBranch.country.carTax
```

```
context   LateReturn:: extraInterval() : Duration
post:
          result= self.actualReturn-self.ending
```

EarlyReturn

```
context   EarlyReturn:: onRentInterval() :     Duration
post:
          result= self.actualReturn-self.beginning


context   EarlyReturn:: allInstances() : Set(EarlyReturn)
post:
          ClosedRental.allInstances()->select(initEnding-
            actualReturn> hour(1))
```

BadExperience

```
context   BadExperience:: typeIsKey() :  DateTime
post:
          result=BadExperience.allInstances()->select(b|
            b.name=self.name)->size()=1
```

CarDamage

```
context   CarDamage:: allInstances() :   Set(CarDamage)
post:
          result=BadExperience.allInstances()->
            select(b|b.type=BadExpType::carDamage)
```

AssignedCar

```
context   AssignedCar:: pickUpBbranchIsResponsible() :     Boolean
post:
          result= (self.RentalAgreement.oclIsTypeOf(OpenedRental) and
            not(oclIsTypeOf(ClosedRental)) implies
          self.car.branch= self.rentalAgreement.pickUpBranch
documentation:
          When a car is assigned to a reservation, the pick-up
          branch is resposible for it
```

Car

```
context   Car:: onlyOneAssignment() :    Boolean
post:
          result= car.rentalAgreement->select(rA |
            not(rA.oclIsTypeOf(CanceledReservation) and
            not(rA.oclIsTypeOf(ClosedRental)))->size()<=1
documentation:
          A car can only be assigned to a reservation in a
          certain date.
```

```
context   Car:: registrationNumberIsKey() :    Boolean
post:
          result=Car.allInstances()->select(b|b.registrationNumber=
            self.registrationNumber)->size()=1


context   Car:: carGroup() :  CarGroup
post:
          result=self.carModel.carGroup
```

OwnCar

```
context   OwnCar:: available?() :  Boolean
post:
          result= not(self.oclIsTypeOf(NeedsMaintenance)) and
          not(self.oclIsTypeOf(RepairsScheduled)) and
          not(self.oclIsKindOf(ToBeSoldCar)) and not(self.assigned?)
          and not(self.oclIsTypeOf(BeingTransferredCar)) and
          not(self.oclIsTypeOf(NeedToBeSoldCar))


context   OwnCar:: assigned?() :   Boolean
post:
          result= car.rentalAgreement->exists(rA |
            not(rA.oclIsTypeOf(CanceledReservation) and
            not(rA.oclIsTypeOf(ClosedRental) ) )
```

NeedToBeSoldCar

```
context   NeedToBeSoldCar:: allInstances() :   Set(NeedToBeSoldCar)
post:
          OwnCar.allInstances()->select(c|today()-c.acquisitionDate>=
            year(1) or self.currentMileage>=40,000)
```

NeedsMaintenance

```
context   NeedsMaintenance:: notOver10PerCent() :    Boolean
post:
          result= currentMileage -mileageFromLastService
          <=1,1*mileageForService or Now() - lastMaintenanceDate
          <= 1,1*timeForService


context   NeedsMaintenance:: allInstances() :  Set(NeedsMaintenance)
post:
          result= OwnCar.allInstances()->select(currentMileage
          -mileageFromLastService >=
          MaintenanceRequirements.mileageForService or Now() –
          lastMaintenanceDate > MaintenanceRequirements.timeForService)
documentation:
          A car needs maintenance if it was serviced more than 3
          months ago or has accumulated more than 10,000 km
          since the last service.
```

ToBeSoldCar

```
context   ToBeSoldCar:: notAssignedToReservation() : Boolean
post:
          self.rentalAgreement->forAll(r| r.oclIsKindOf(ClosedRental) or
          r.oclIsKindOf(CanceledReservation))
```

MaintenanceScheduled

```
context   MaintenanceScheduled:: serviceDepot() :    ServiceDepot
post:
          let sd:ServiceDepot=self.Branch.ServiceDepot
          in
          let occupation:Natural=sd.MaintenanceScheduled->
          select(ms|ms.beginningDate=self.beginningDate)->size()
          in
          capacity>occupation implies result=sd
```

ServiceDepot

```
context   ServiceDepot:: nameIsKey() :   Boolean
post:
          result=ServiceDepot.allInstances()->
            select(s|s.name=self.name)->size()=1
```

CarModel

```
context   CarModel:: nameIsKey() : Boolean
post:
          result=CarModel.allInstances->
            select(b|b.name=self.name)->size()=1
```

CarGroup

```
context   CarGroup:: nameIsKey() : Boolean
post:
          result=CarGroup.allInstances()->
            select(b|b.name=self.name)->size()=1


context   CarGroup:: totalOrder() :      Boolean
post:
          let isWorse(w,b:CarGroup):Boolean= b.worse=w or
isWorse(w,b.worse)
          let isBetter(b,w:CarGroup):Boolean= w.better=b or
            isBetter(b,w.better)
          in
          result = CarGroup.allInstances()->one(cg|cg.worse->isEmpty())
          and CarGroup.allInstances()->one(cg|cg.better->isEmpty()) and
          CarGroup.allInstances()->forall(cg1,cg2|  isWorse(cg1,cg2)
          implies not isBetter(cg1,cg2) and isBetter(cg1,cg2) implies
          not isWorse(cg1,cg2))


context   CarGroup:: quotaForAllBranches() :   Boolean
post:
          result=self.carGroupQuota->size()=Branch.allInstances()->size()
```

PendantCarOrder

```
context   PendantCarOrder:: idIsKey() :  Boolean
post:
          result=PendantCarOrder.allInstances()->
            select(b|b.id=self.id)->size()=1
```

EU_RentPerson

```
context   EU_RentPerson:: is25OrOlder() :      Boolean
post:
          result= (now()- self.birthdate) >=year(25)
documentation:
          Eu-rent persons must be 25 years old or older.


context   EU_RentPerson:: idIsKey() :    Boolean
post:
          result=EU_RentPerson.allInstances()
            ->select(p|p.id=self.id)->size()=1


context   EU_RentPerson:: faults() :     set(FaultSeriousness)
post:
          let faultsAsDriver: FaultSeriousness= self.rentalsAsDriver->
            select(rA|   rA.oclIsTypeOf(ClosedRental)).faultSeriousness
          let faultsAsRenter: FaultSeriousness= self.RentalAgreement->
            select(rA|   rA.oclIsTypeOf(ClosedRental)).faultSeriousness
          in
            result= self.faultsAsDriver->oclAsType(Set)->
              union(self.faultsAsRenter)->asSet()


context   EU_RentPerson:: belongingBranch() :  Branch
post:
          let firstRental:RentalAgreement=self.rentalsAsDriver->
          union(self.rentalsAsRenter)->sortedBy(beginning)->first()
          in
          result=firsRental.pickUpBranch
```

Customer

```
context   Customer:: rentalsDoNotOverlap() :   Boolean
post:
          result=self.RentalAgreement-> reject(rA|
            rA.oclIsKindOf(CanceledReservation)->notExists(rA |
            self.rentalAgreement->select(rAOther | rAOther.beginning.day()>
            rA.beginning.day())->exists(rAOther| rAOther.beginning.day() <=
            rA.agreedEnding.day()))
documentation:
          A customer's rental periods cannot overlap
```

LoyaltyMember

```
context   LoyaltyMember:: availablePoints() :  Natural
post:
          let candidateRentals: Set(ClosedRental)= self.RentalAgreement->
            select(rA| rA.oclIsTypeOf(ClosedRental) and (now()- rA.ending)<
            year(3) and rA.ending > (membershipDate - year(1))->
            oclAsType(Set(ClosedRental))
          let earnRentals: Set(ClosedRental)= candidateRentals->
            reject(cR|cR.oclIsTypeOf(PaidWithPointsRental)
          let accumulatedPoints: Integer= earnRentals->forAll(r |
            result->including(pointsEarned(r.bestPrice)))->sum()
          let spendRentals: Set(ClosedRental)=
          candidateRental->select(oclIsTypeOf(PaidWithPointsRental))
          let spentPoints: Integer= spendRentals->forAll(r |
            result->including  (pointsSpent(r.bestPrice)))->sum()
          in
          result= accumulatedPoints-spentPoints


context   LoyaltyMember:: meetsLoyalPermanence() :   Boolean
post:
          result= self.RentalAgreement.beginning->exists(dT| dT>
          (now()-year(1))) and self.faults->isEmpty()
documentation:
          A member of the loyalty incentive scheme has done at
          least a rental during a year and has not recorded any
          bad experience.
```

Blacklisted

```
context   Blacklisted:: noRentals() :    Boolean
post:
          result= self.rentalsAsDriver->select(rA| rA.beginning
            > self.blacklistedDate)->
            forAll(rA|ra.oclIsTypeOf(CanceledReservation))
documentation:
          Once an EU-rent person has been blacklisted, cannot
          participate in a rental or make a rental.
```

DrivingLicense

```
context   DrivingLicense:: validLicense() :    Boolean
post:
          result=today()-self.issue>year(1) and
          self.eu_RentPerson.rentalsAsDriver.agreeEnding->
          forAll(d|d<self.expiration)


context   DrivingLicense:: numberIsKey() :     Boolean
post:
          result=DrivingLicense.allInstances()->
            select(d|d.number=self.number)->size()=1
```

RentalDuration

```
context   RentalDuration:: nameIsKey() : Boolean
post:
          result=RentalDuration.allInstances()->
            select(b|b.name=self.name)->size()=1


context   RentalDuration:: coherentPrices() :  Boolean
post:
          let curCGDPrices: Set(CarGroupDurationPrice) =
            self.carGroupDurationPrice->reject(cgdp|
            cgdp.oclIsTypeOf(EndDurationPrice))
          in
          result = curCGDPrices->forAll(cgdp|cgdp.price>=
          curCGDPrices.CarGroup.worse.CarGroupDurationPrice->
          select(cg|cg.RentalDuration=self).price)


context   RentalDuration:: totalOrder() :      Boolean
post:
          let isShorter(s,l:RentalDuration):Boolean= l.shorter=s or
             isShorter(s,l.shorter)
          let isLonger(l,s:RentalDuration):Boolean= s.longer=l or
             isLonger(l,s.longer)
          in
            result = RentalDuration.allInstances()->one(rd|
            rd.shorter->isEmpty()) and RentalDuration.allInstances()->
            one(rd|rd.longer->isEmpty()) and RentalDuration.allInstances()
            ->forAll(rd1,rd2| isShorter(rd1,rd2) implies not
            isLonger(rd1,rd2) and isLonger(rd1,rd2) implies not
            isShorter(rd1,rd2))
```

Discount

```
context   Discount:: nameIsKey() : Boolean
post:
          result=Discount.allInstances()->
            select(b|b.name=self.name)->size()=1
```

ClosedDiscount

```
context   ClosedDiscount:: correctEnding() :   Boolean
post:
          result= self.endingDate >= self.beginningDate
```

EndDurationPrice

```
context   EndDurationPrice:: correctEnding() : Boolean
post:
          result= self.beginning <= self.endingDate
```

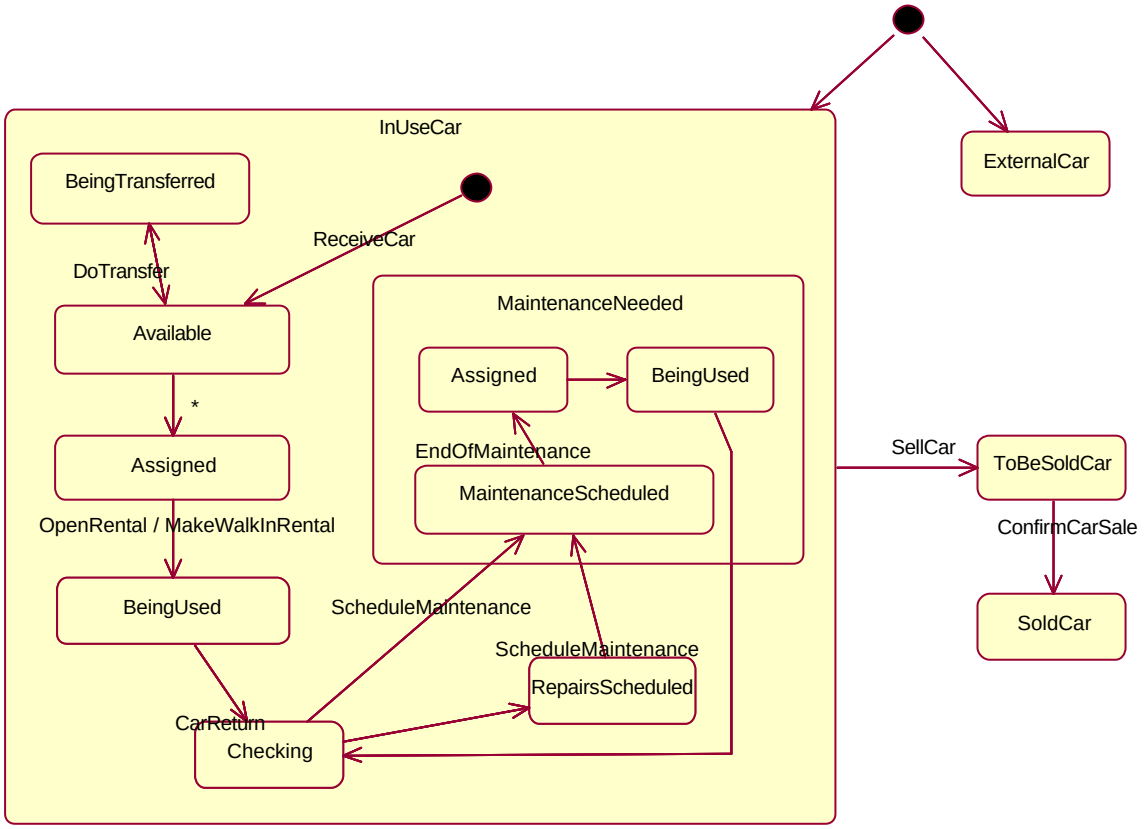
6. STATE MODEL

Overview

State diagrams are used to clarify the acceptable transitions between the states. Although it is not a crucial part in the development of this project, it has been considered appropriate and clarifying to define the state diagrams for some of the main entities of the system.

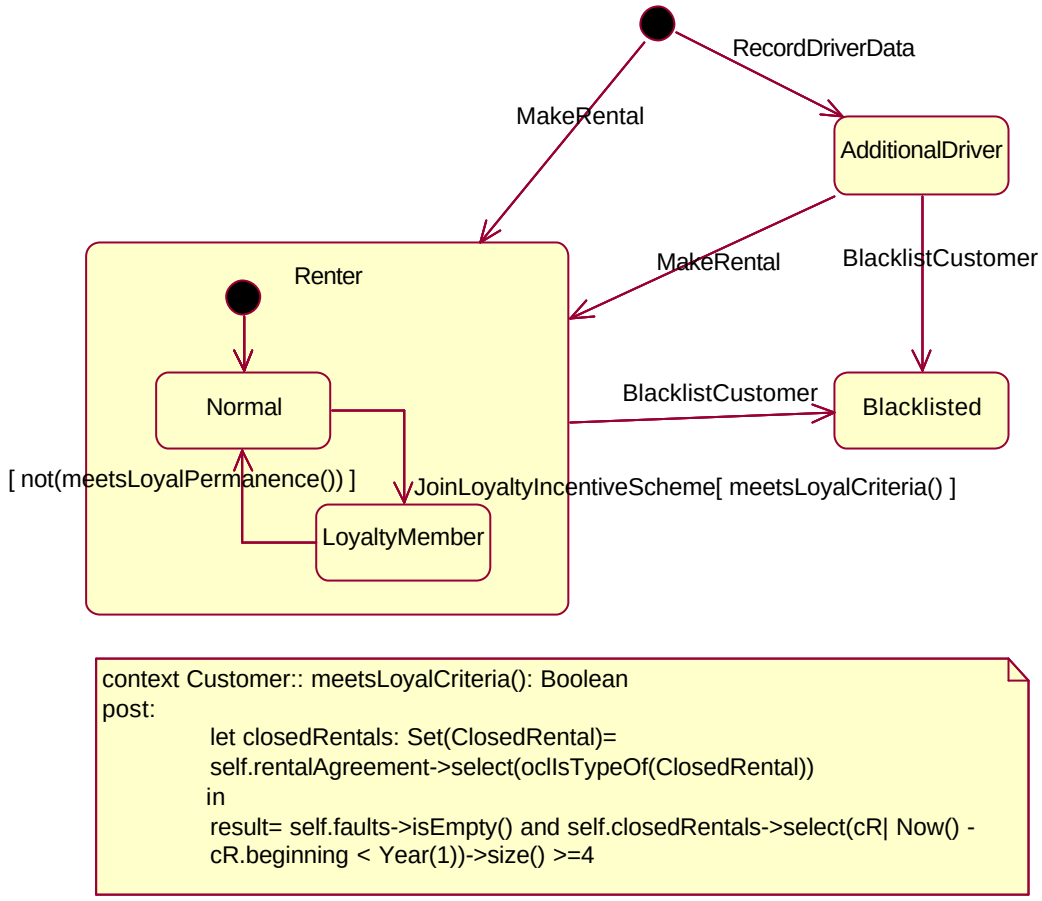
Diagrams

Car



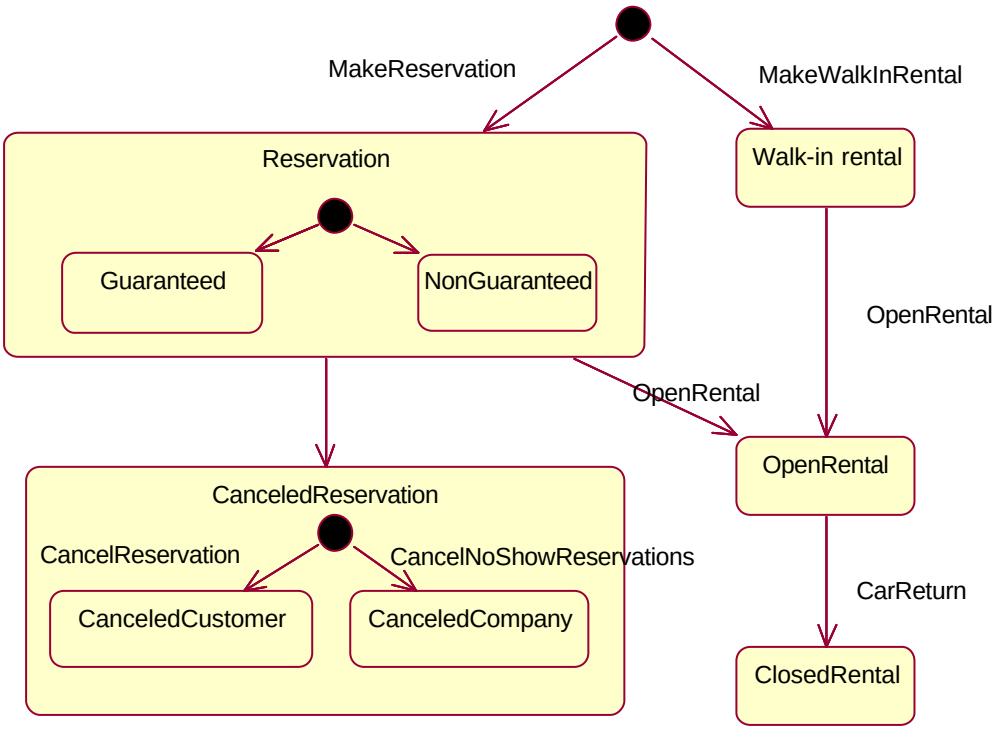
* CarAllocationAutomatic / CarAllocationExceptionOption / CarAllocationExtremisOption

EU_RentPerson




```
context Customer:: meetsLoyalCriteria(): Boolean
post:
        let closedRentals: Set(ClosedRental)=
        self.rentalAgreement->select(oclIsTypeOf(ClosedRental))
       in
        result= self.faults->isEmpty() and self.closedRentals->select(cR| Now() -
        cR.beginning < Year(1))->size() >=4
```

RentalAgreement



7. EVENTS MODELLING

Overview

Once the system use cases have been defined, system events are obtained analyzing carefully and identifying the interactions between the system and the actors. The typical approach to model events is via the so-called *system operations.*

However, in this document a different approach has been aimed to be tested. This approach consists in modelling the events as objects as explained in [EE] and so, exploit the characteristics of the OO to avoid, for example, the rewriting of the same constraint in different events.

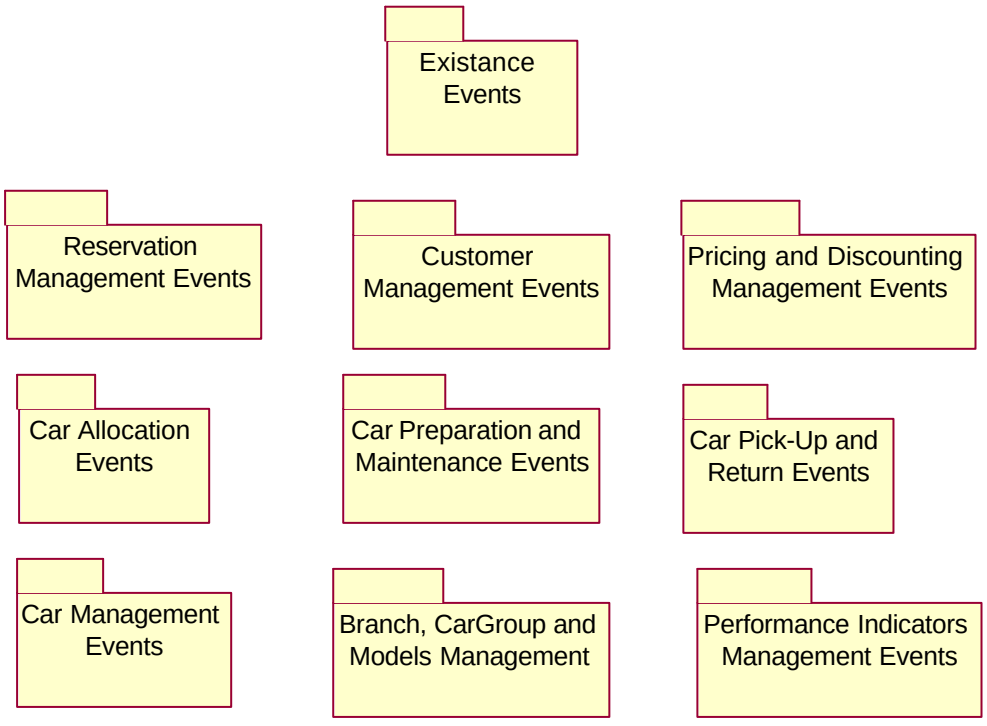
This technique has been combined with the already introduced ones to specify constraints and derivation rules and so, being consistent with the static model.

Previous remarks

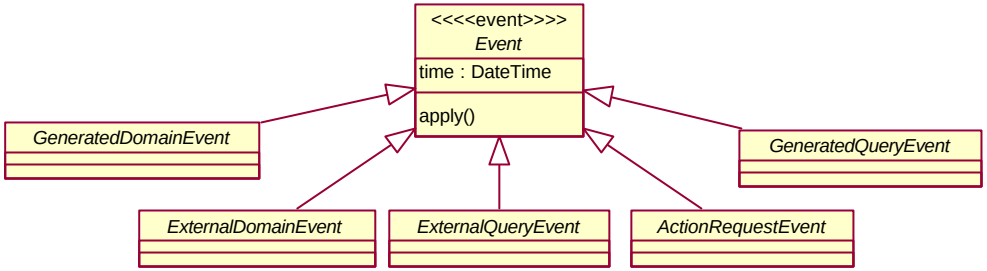
In EU-Rent Rentals case there are some events of a considerable difficulty and importance. For convenience, this events have been split up in smaller events to favour understanding and self-description, because otherwise would be almost impossible to describe some of the events formally.

Event diagrams

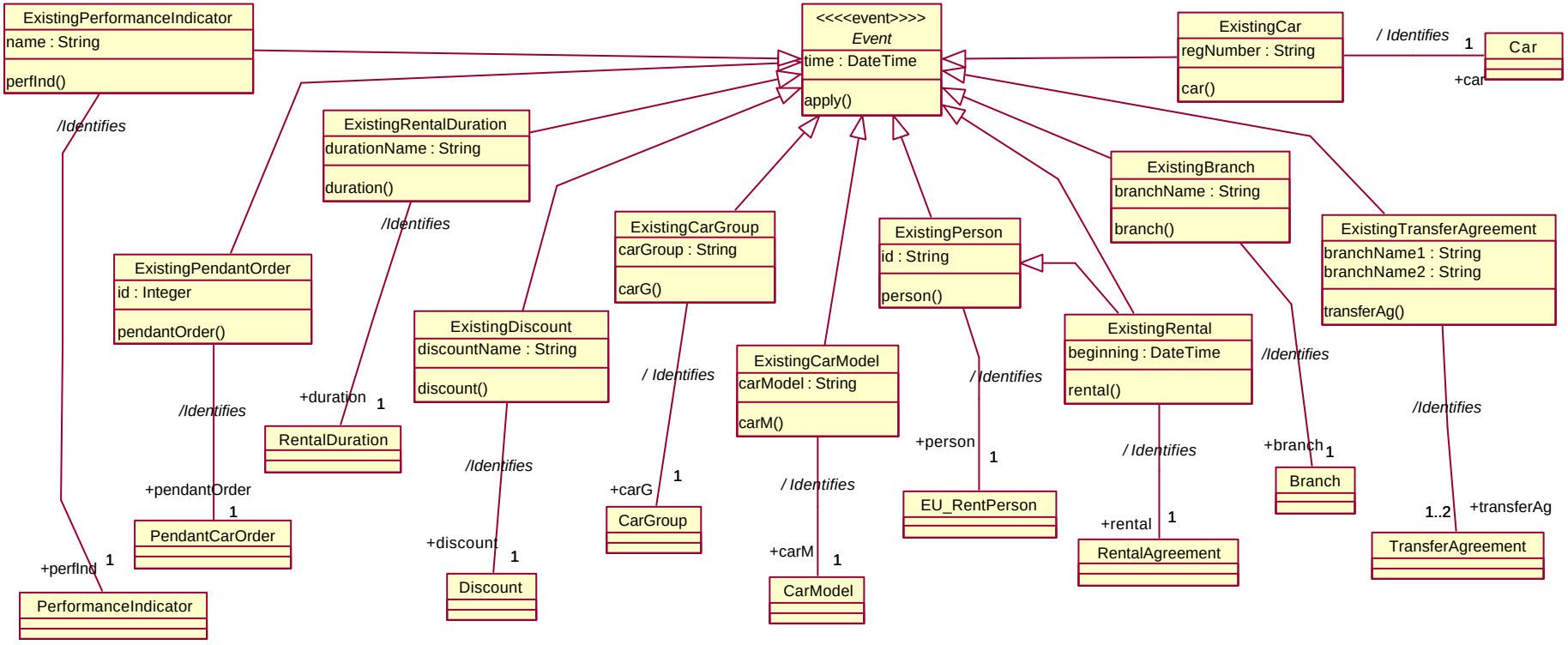
The events modelling has been divided in packages, corresponding to thematic areas, in order to ease understanding. These are the following:



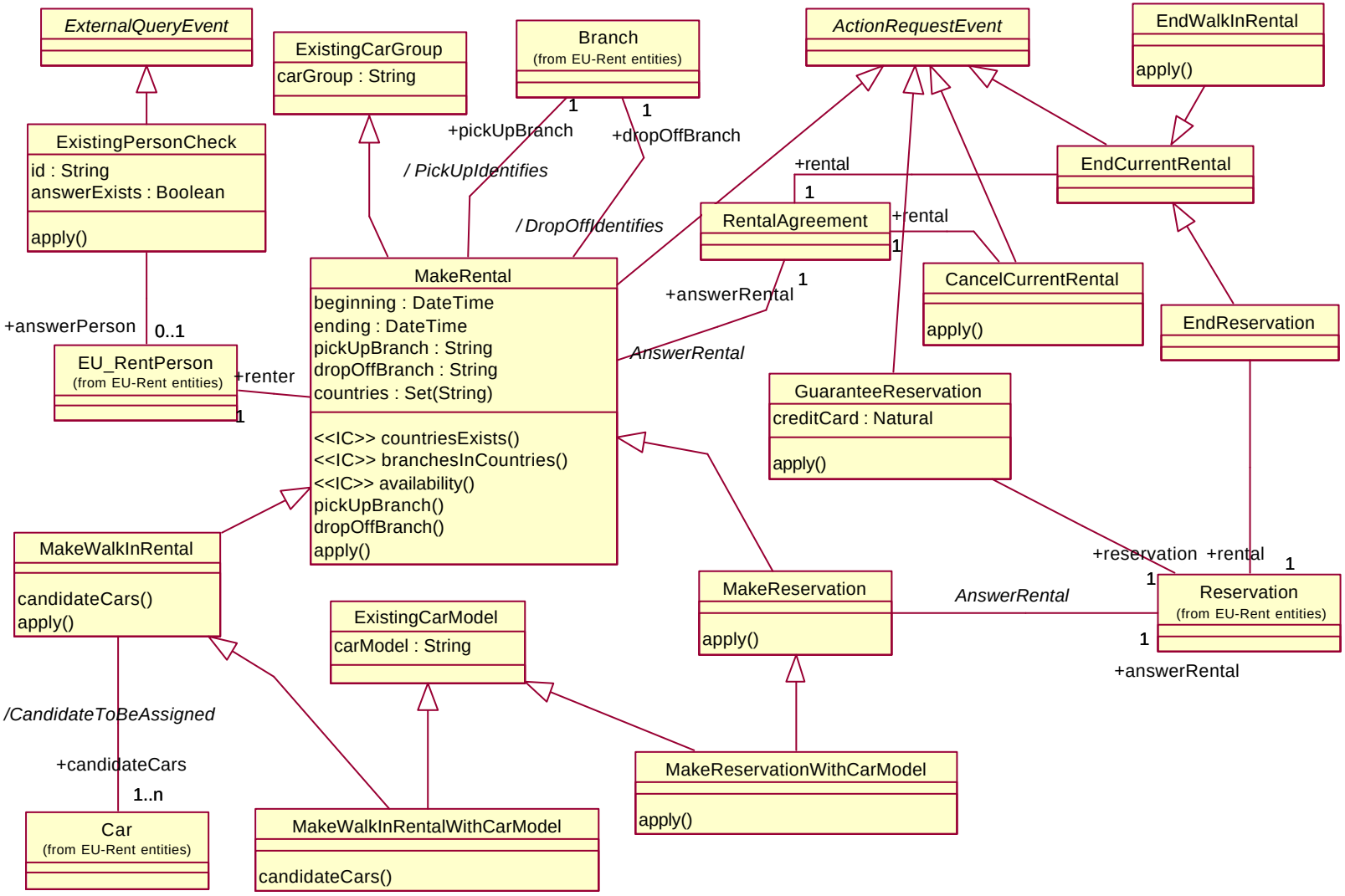
Events hierarchy

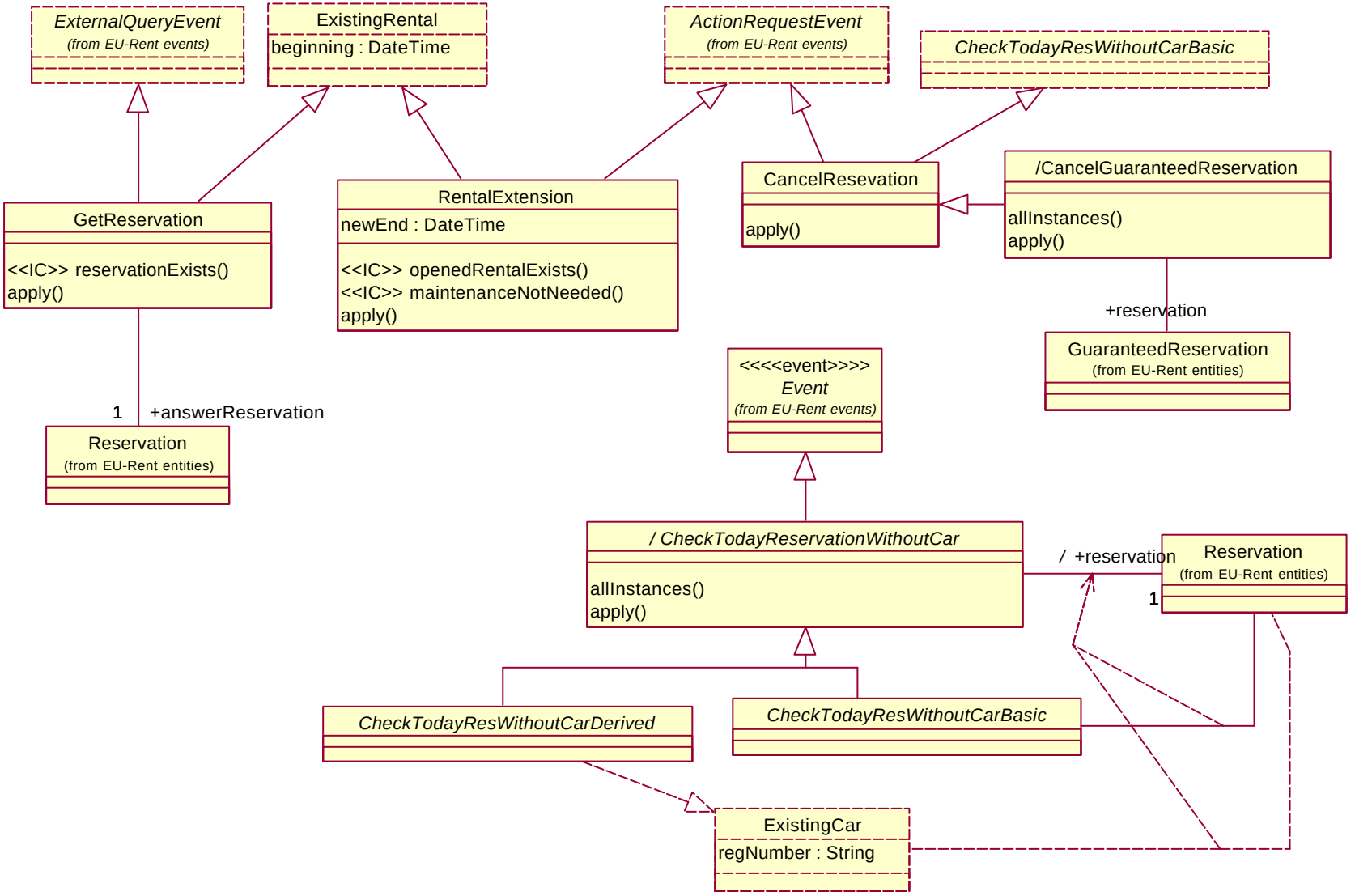


Existance Events

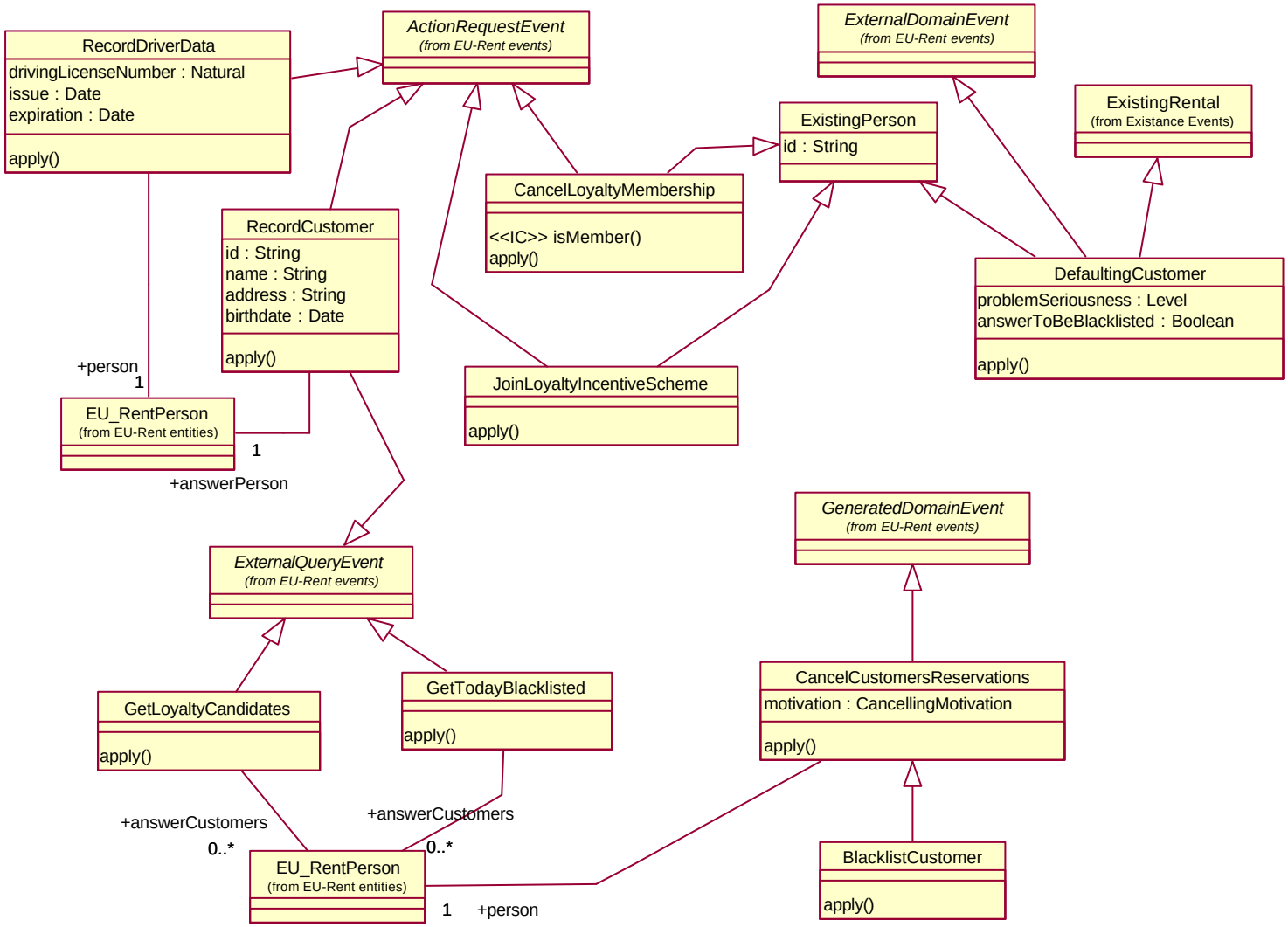


Reservation Management Events

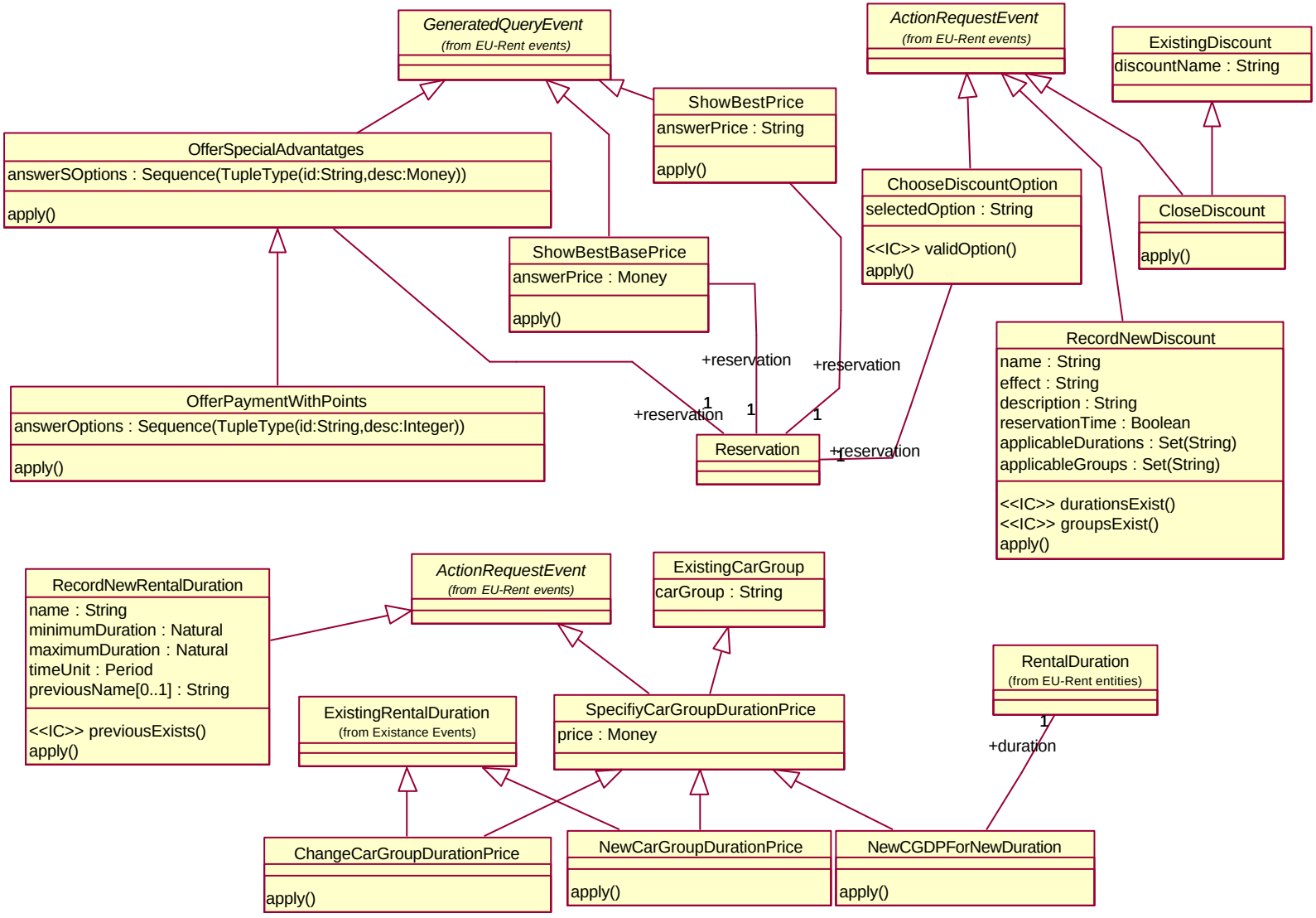



ExternalQueryEvent
(from EU-Rent events)
ExistingRental
beginning : DateTime
ActionRequestEvent
(from EU-Rent events)
CheckTodayResWithoutCarBasic
GetReservation
<<IC>> reservationExists()
apply()
RentalExtension
newEnd : DateTime
<<IC>> openedRentalExists()
<<IC>> maintenanceNotNeeded()
apply()
CancelResevation
apply()
/CancelGuaranteedReservation
allInstances()
apply()
+reservation
GuaranteedReservation
(from EU-Rent entities)
1
+answerReservation
Reservation
(from EU-Rent entities)
<<<<event>>>>
Event
(from EU-Rent events)
/ CheckTodayReservationWithoutCar
allInstances()
apply()
/ +reservation
Reservation
(from EU-Rent entities)
1
CheckTodayResWithoutCarDerived
CheckTodayResWithoutCarBasic
ExistingCar
regNumber : String

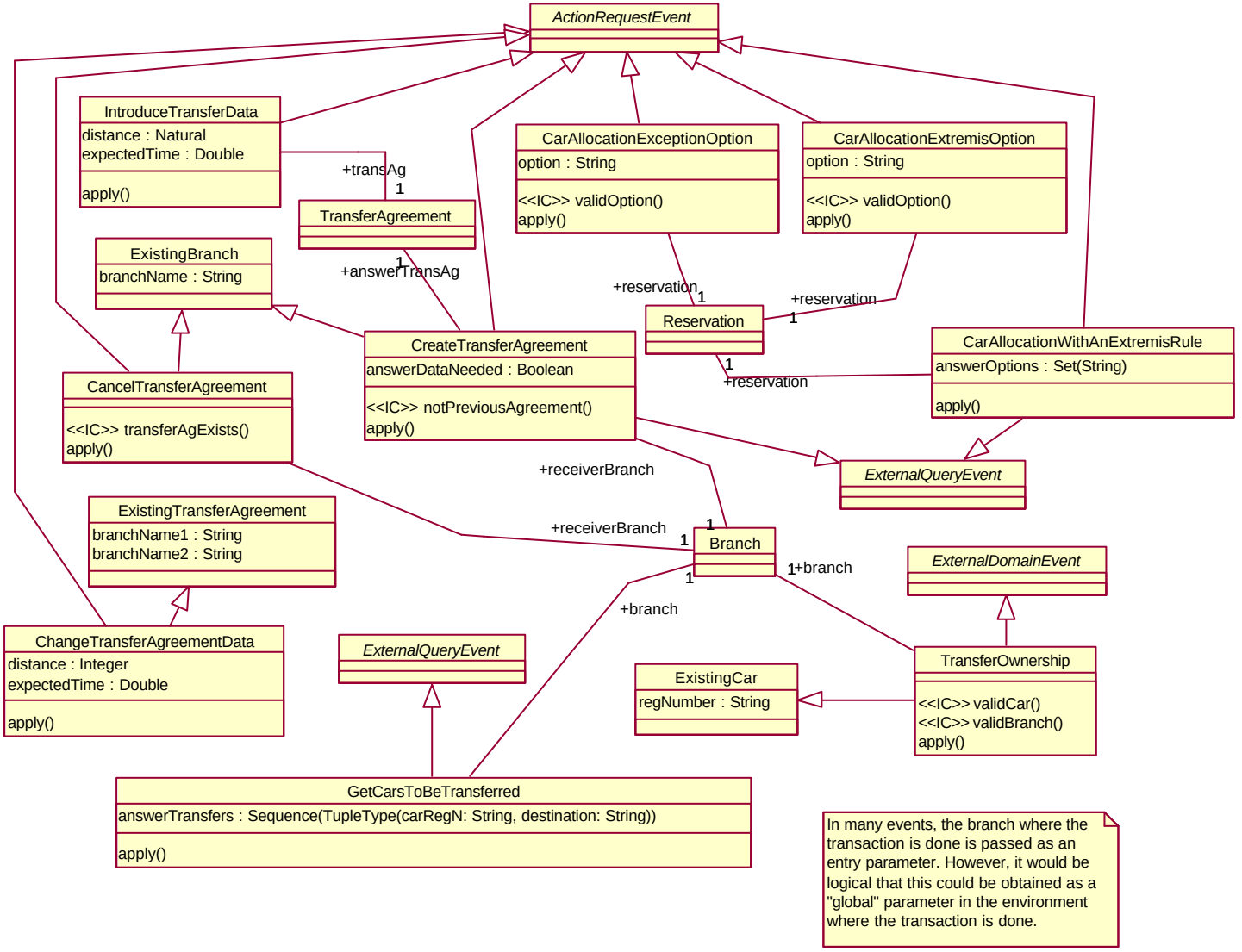
Customer Management Events

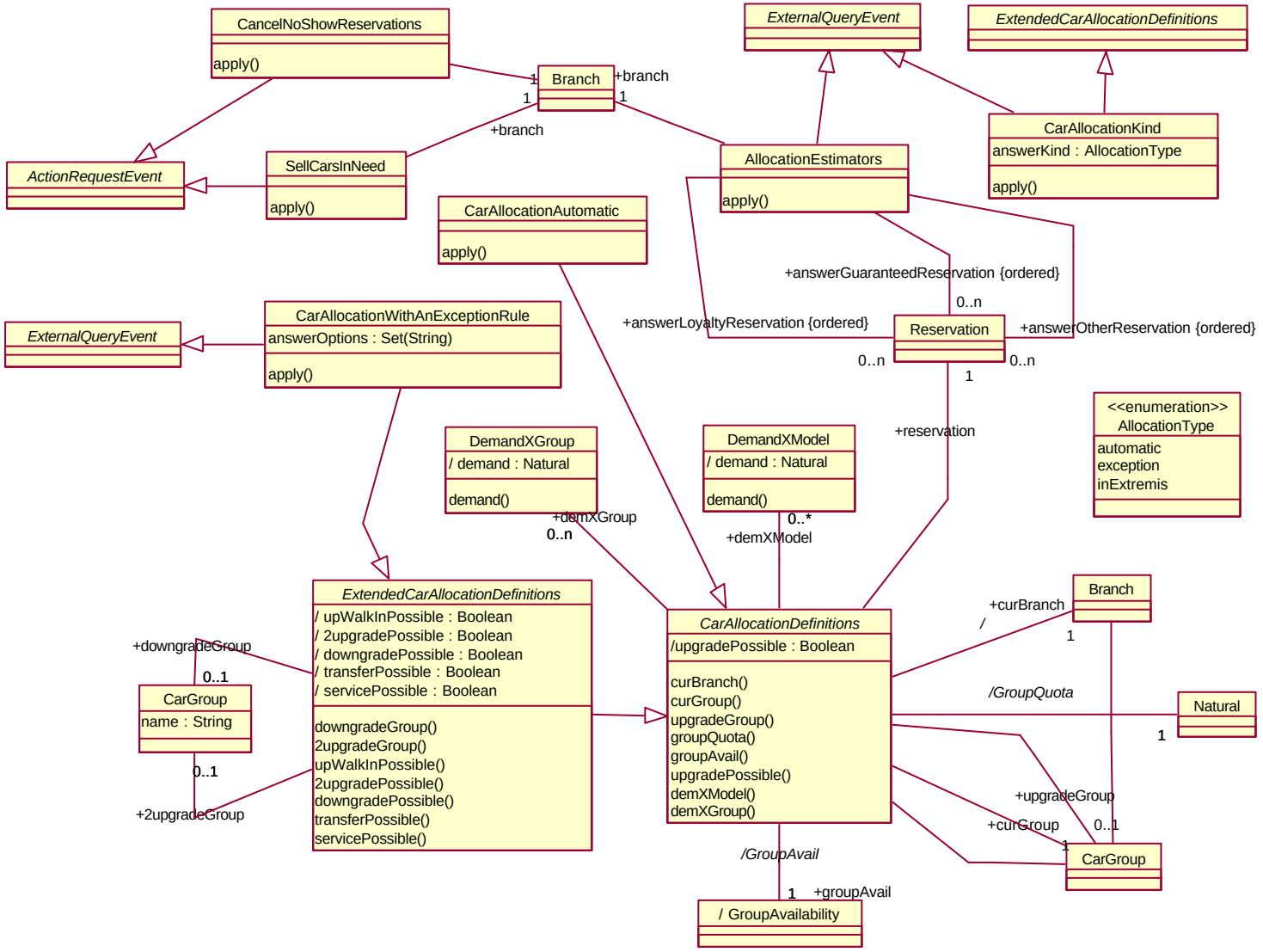


Pricing and Discounting Management Events

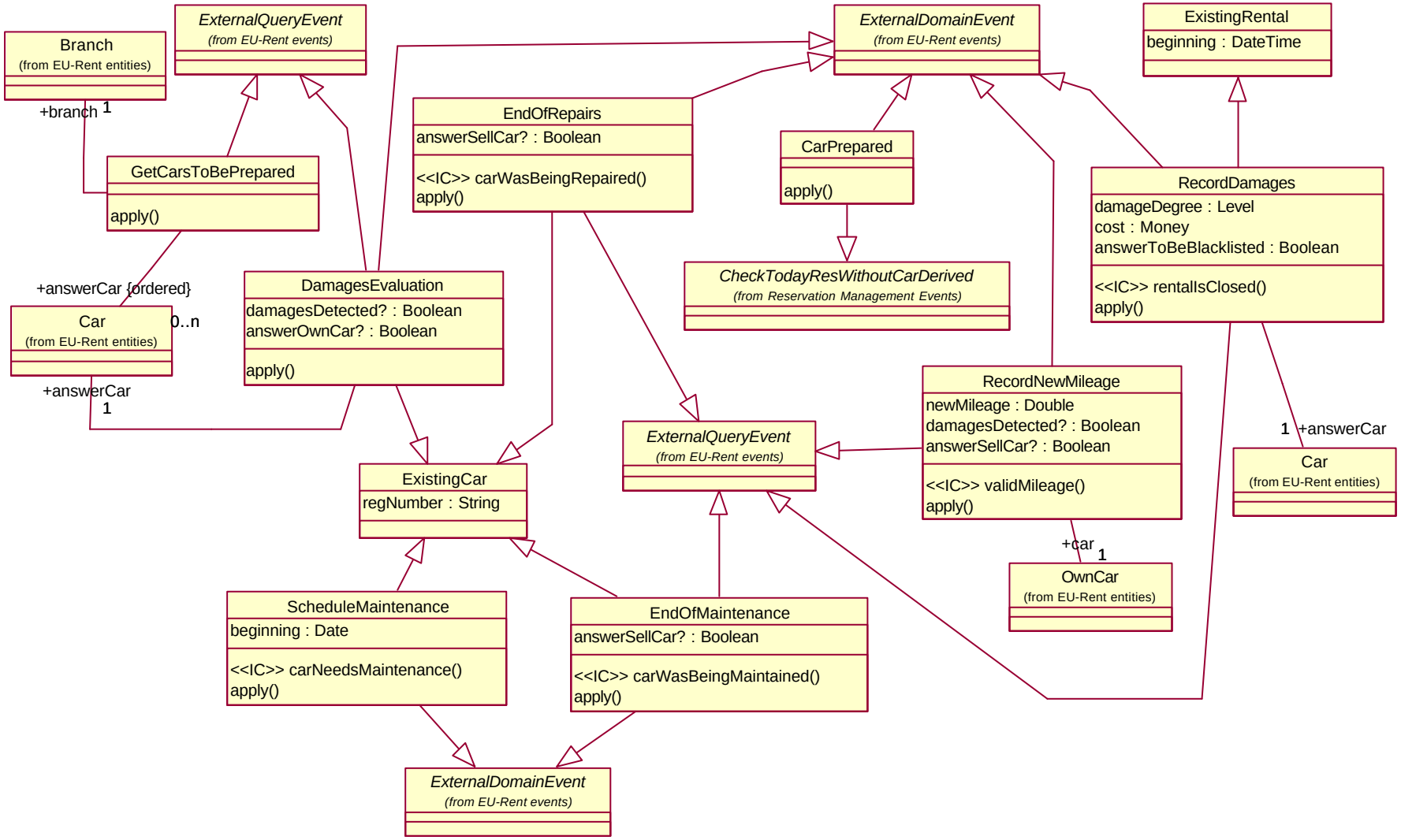


Car Allocation Events

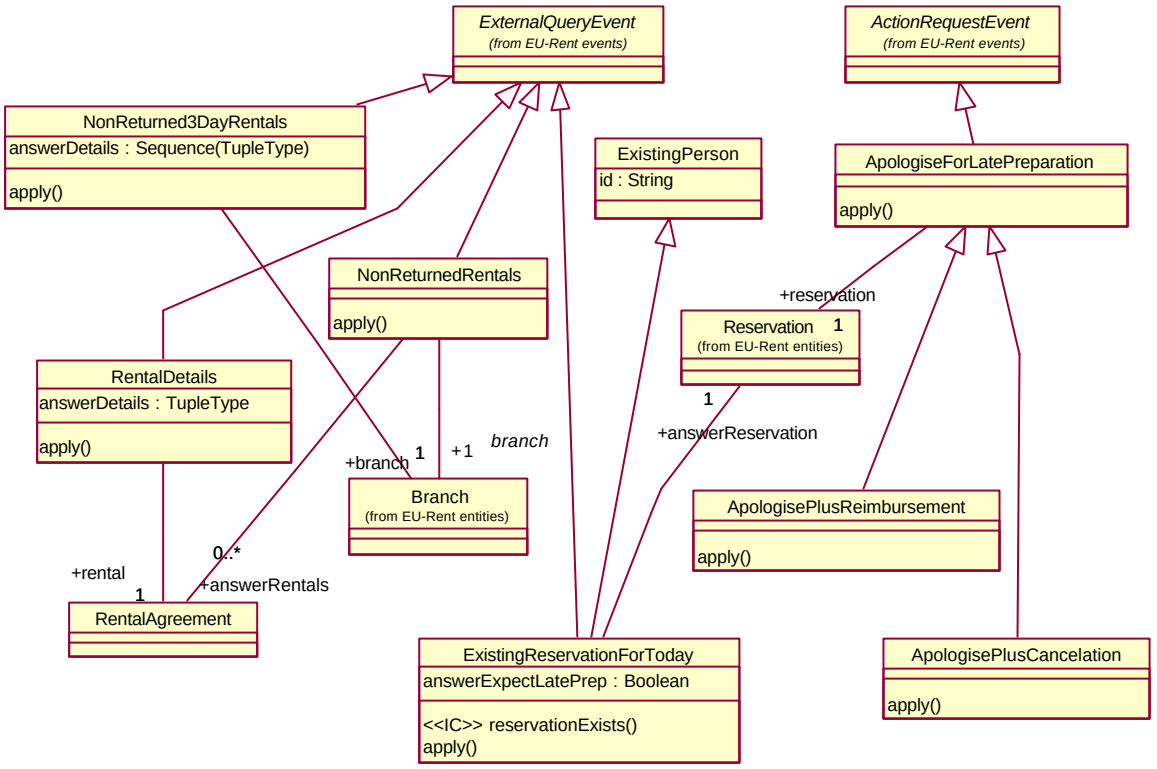


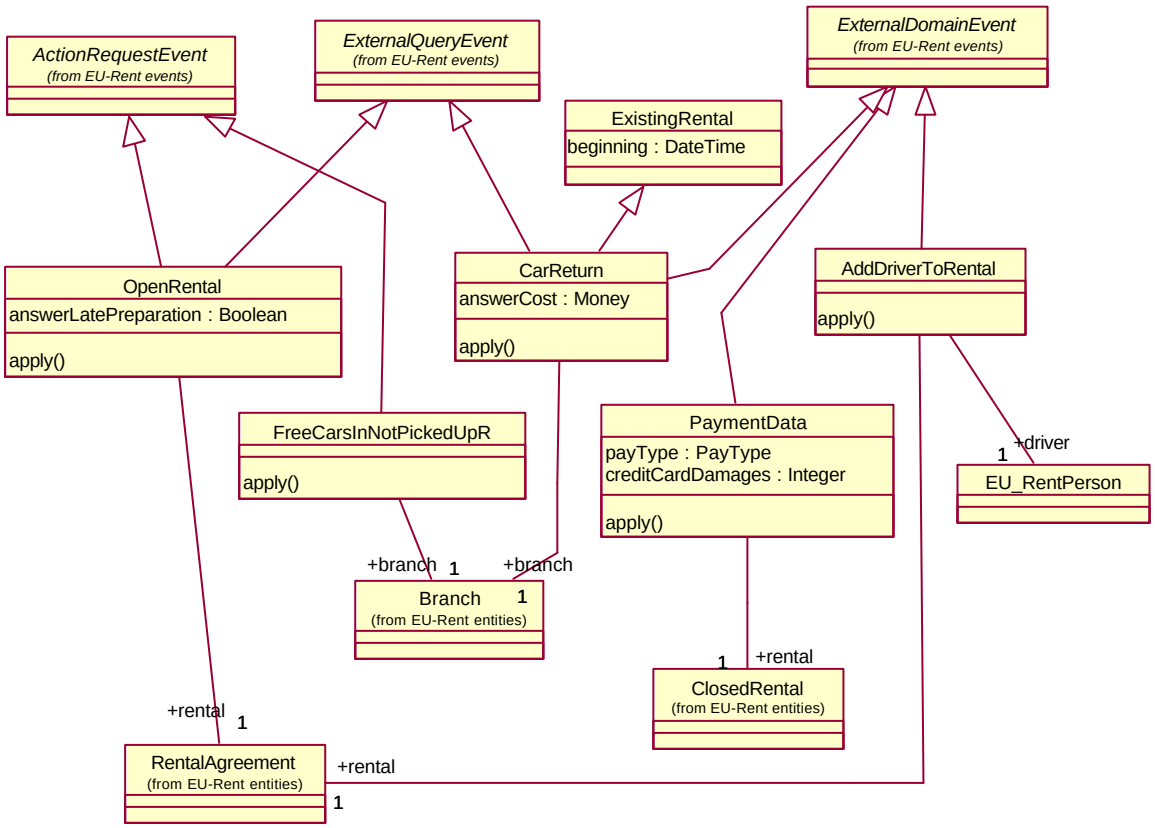

CancelNoShowReservations
apply()
Branch
+branch
1
1
+branch
1
ExternalQueryEvent
ExtendedCarAllocationDefinitions
CarAllocationKind
answerKind : AllocationType
apply()
ActionRequestEvent
SellCarsInNeed
apply()
AllocationEstimators
apply()
CarAllocationAutomatic
apply()
+answerGuaranteedReservation {ordered}
0..n
+answerLoyaltyReservation {ordered}
Reservation
+answerOtherReservation {ordered}
0..n
0..n
1
ExternalQueryEvent
CarAllocationWithAnExceptionRule
answerOptions : Set(String)
apply()
+reservation
<<enumeration>>
AllocationType
automatic
exception
inExtremis
DemandXGroup
/ demand : Natural
demand()
DemandXModel
/ demand : Natural
demand()
+demXGroup
0..n
0..*
+demXModel
ExtendedCarAllocationDefinitions
/ upWalkInPossible : Boolean
/ 2upgradePossible : Boolean
/ downgradePossible : Boolean
/ transferPossible : Boolean
/ servicePossible : Boolean
downgradeGroup()
2upgradeGroup()
upWalkInPossible()
2upgradePossible()
downgradePossible()
transferPossible()
servicePossible()
CarAllocationDefinitions
/upgradePossible : Boolean
curBranch()
curGroup()
upgradeGroup()
groupQuota()
groupAvail()
upgradePossible()
demXModel()
demXGroup()
+curBranch
/
1
Branch
/GroupQuota
Natural
1
+downgradeGroup
0..1
CarGroup
name : String
0..1
+2upgradeGroup
+upgradeGroup
+curGroup
0..1
1
CarGroup
/GroupAvail
1
+groupAvail
/ GroupAvailability

Car Preparation and Maintenance Events

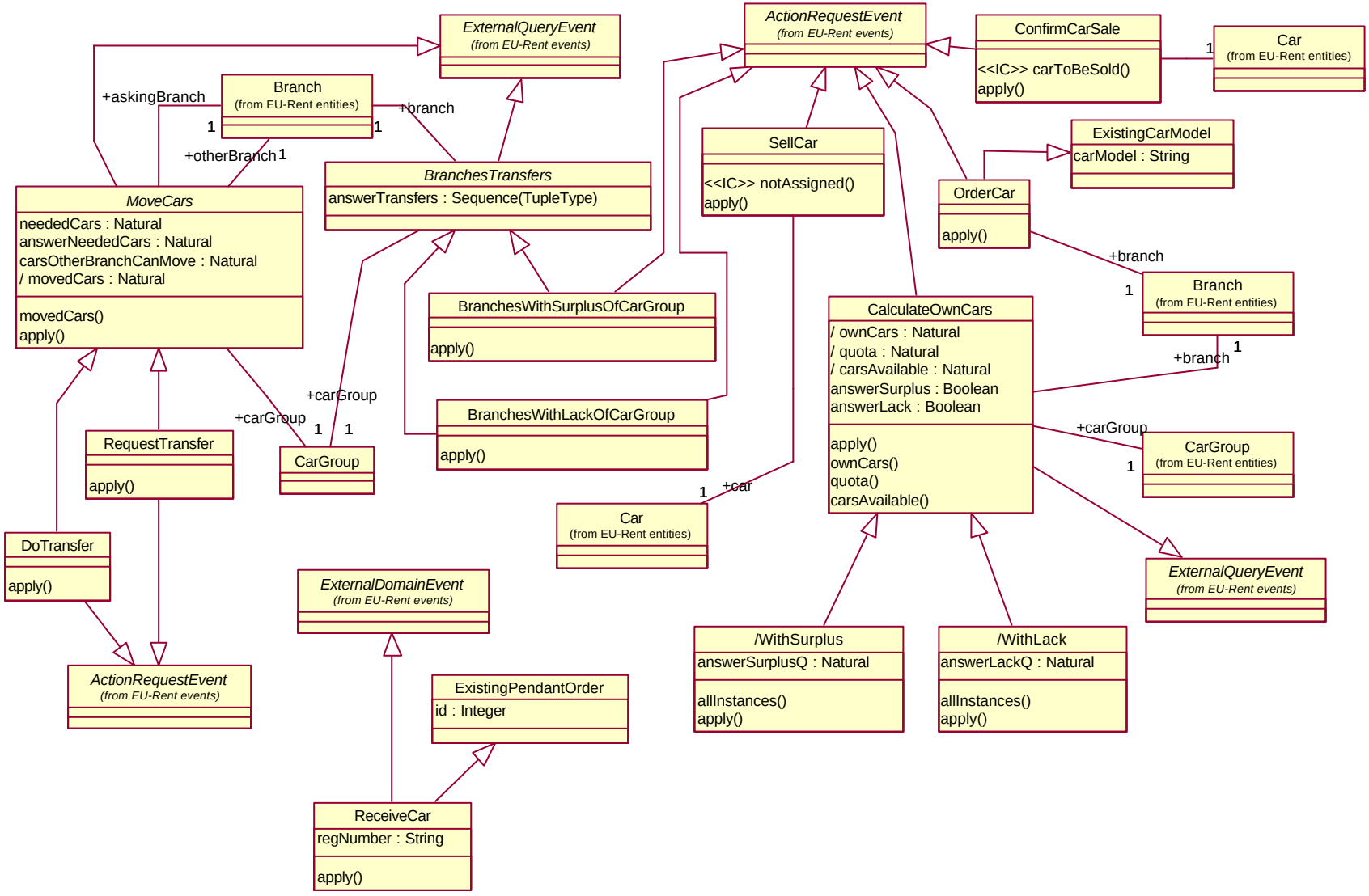


Car Pick-up and Return Events

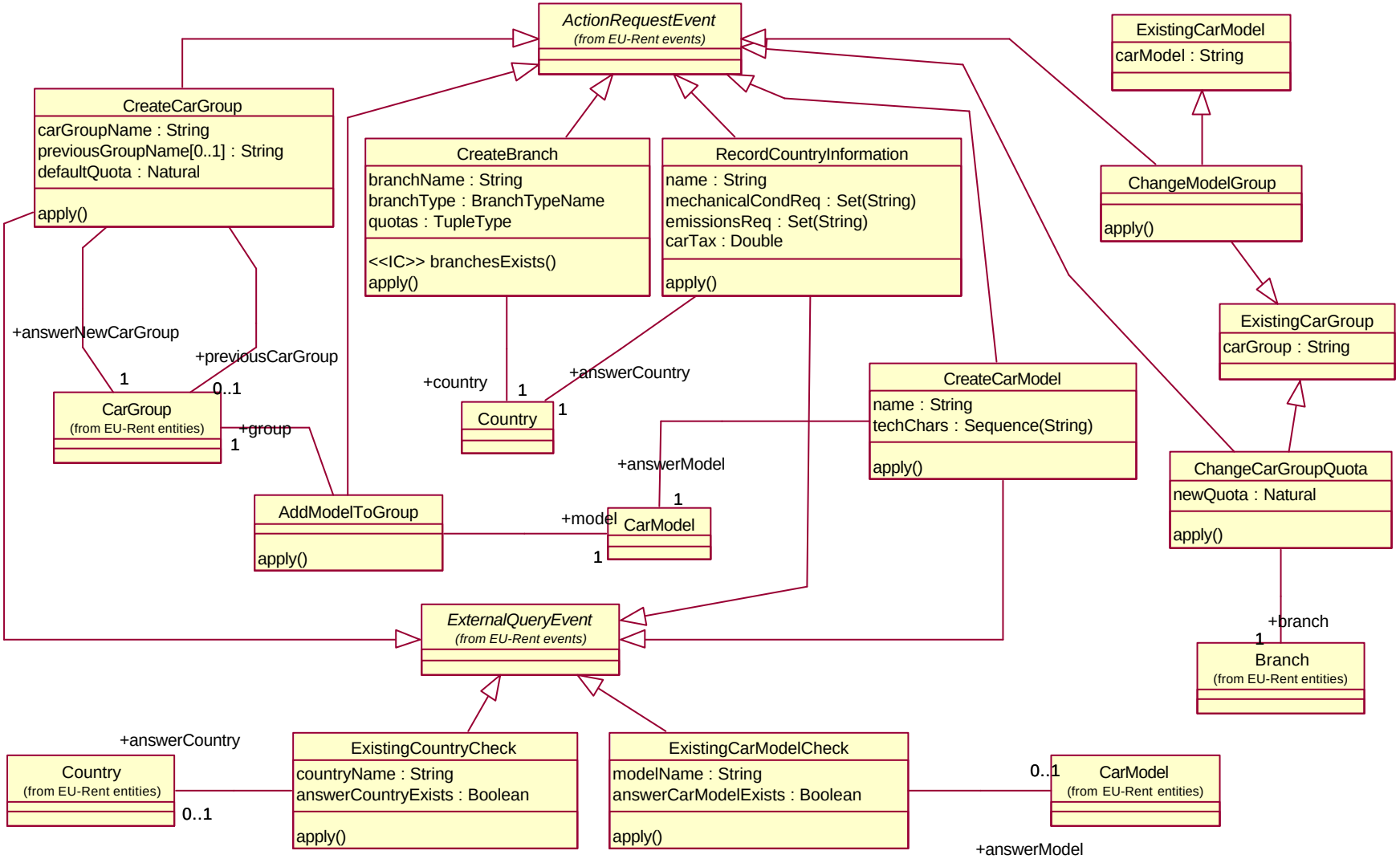




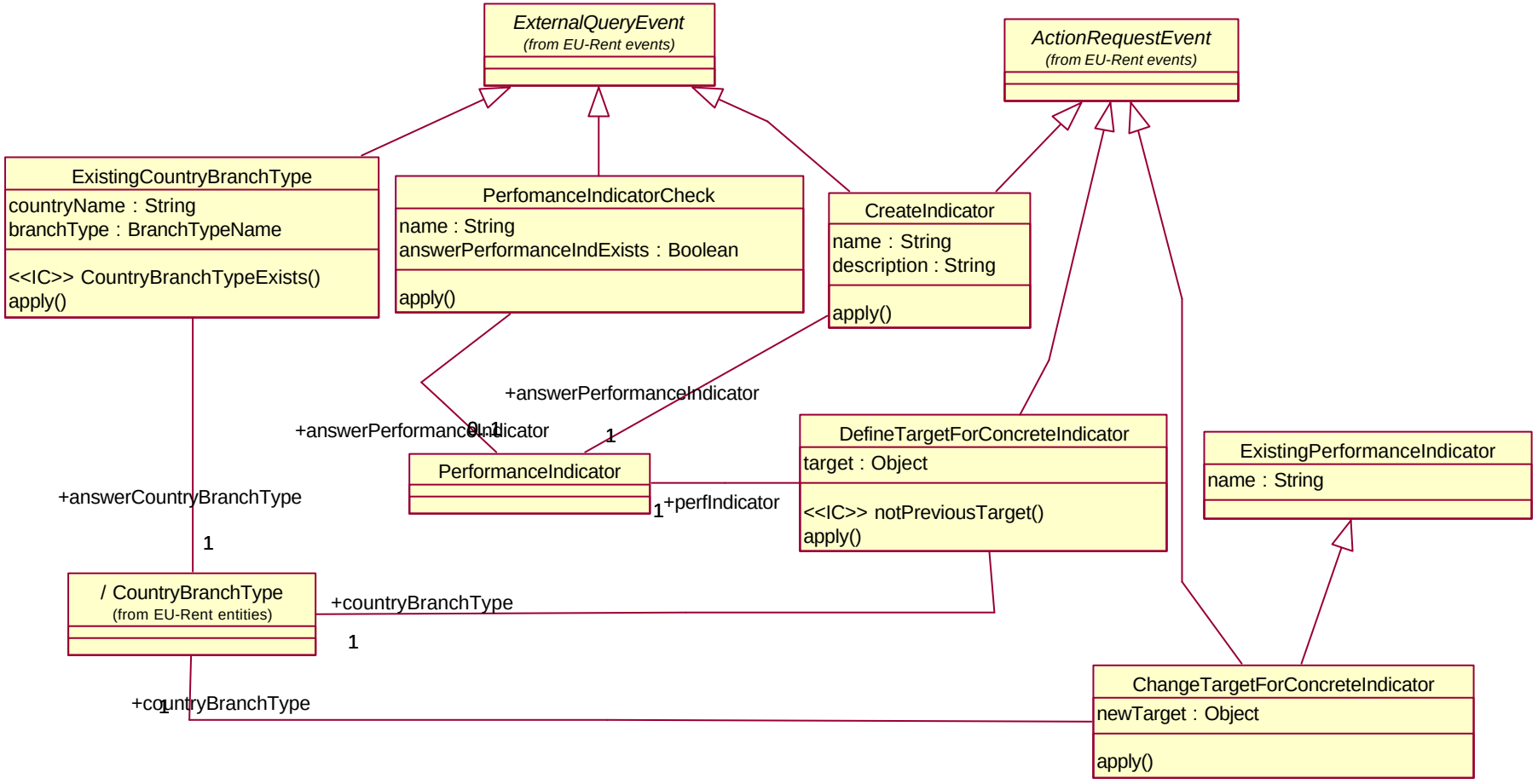
Car Management Events



Branch, Car Group and Models Management Events



Performance Indicators Events



Complete specification of defining event operations and their auxiliary associated to derived elements and integrity constraints

EXISTANCE EVENTS

ExistingBranch

```
context   ExistingBranch:: branch() :    Branch
post:
          let br:Set(Branch)=Branch.allInstances()->
          select(b|b.name=self.branchName)
          in
          br->notEmpty() implies result=br->any()
```

ExistingTransferAgreement

```
context   ExistingTransferAgreement:: transferAg() : TransferAgreement
post:
          let transAg: Set(TransferAgreement)=TransferAgreement.
            allInstances()->select(tA|
            (tA.transferor.name=self.branchName1 and tA.receiver.name=
            self.branchName2) or (tA.transferor.name= self.branchName2
            and tA.receiver.name= self.branchName1))
          in
           transAg->isNotEmpty() implies result=transAg->any()
```

ExistingPerson

```
context   ExistingPerson:: person() : Branch
post:
          let euPerson:Set(EU_RentPerson)=
EU_RentPerson.allInstances()
            ->select(p | p.id= self.id)
          in
          euPerson->notEmpty() implies result=euPerson->any()
```

ExistingRental

```
context   ExistingRental:: rental() :    RentalAgreement
post:
          let rent: Set(RentalAgreement)= self.person.RentalAgreement
            ->select(r|r.beginning=self.beginning)
          in
          rent->notEmpty() implies result=rent->any()
```

ExistingCar

```
context   ExistingCar:: car() :    Car
post:
          let carI: Set(Car)=Car.allInstances()->
          select(c|c.registrationNumber=self.regNumber)
          in
```

```
          carI->notEmpty() implies result=carI->any()
```

ExistingCarGroup

```
context   ExistingCarGroup:: carG() :    CarGroup
post:
          let carGr:Set(CarGroup)= carGroup.allInstances()->
          select(cG| cG.name=self.carGroup)
          in
          carGr->notEmpty() implies result=carGr->any()
```

ExistingCarModel

```
context   ExistingCarModel:: carM() :    CarModel
post:
          let carMod: Set(CarModel)=CarModel.allInstances()->
          select(cM| cM.name=self.carModel)
          in
          carMod->notEmpty() implies result=carMod->any()
```

ExistingDiscount

```
context   ExistingDiscount:: discount() :      Discount
post:
          let dis: Set(Discount)= Discount.allInstances()->
            select(d|d.name=self.discountName)
          in
          dis->notEmpty() implies result= dis->any()
```

ExistingRentalDuration

```
context   ExistingRentalDuration:: duration() :      RentalDuration
post:
          let rentDuration:Set (RentalDuration)=RentalDuration.
            allInstances()->select(rd| rd.name=self.durationName)
          in
          rentDuration->notEmpty() implies result= rentDuration->any()
```

ExistingPendantOrder

```
context   ExistingPendantOrder:: pendantOrder() :    PendantCarOrder
post:
          let pendantOrd: Set(pendantCarOrder)=
            pendantCarOrder.allInstances()->select(p| p.id=self.id)
          in
          pendantOrd->size()=1 implies result=pendantOrd->any()
```

ExistingPerformanceIndicator

```
context   ExistingPerformanceIndicator:: perfInd() :
      PerformanceIndicator
post:
          let perf: Set(PerformanceIndicator)= PerformanceIndicator.
          allInstances()->select(pi|pi.name=self.name)
```

```
          in
          perf->notEmpty() implies result=perf->any()
```

RESERVATION MANAGEMENT EVENTS

MakeRental

```
context   MakeRental:: countriesExists() :     Boolean
post:
          self.countries-> forAll(name | Country.allInstances()->
            exists(c|c.name=name))


context   MakeRental:: ranchesInCountries() :  Boolean
post:
          let pickUpCountryN: String= self.pickUpBranch.country.name
          let dropOffCountryN: String= self.dropOffBranch.country.name
          in
          result = self.countries->includes(pickUpCountryN) and
          self.countries->includes(dropOffCountryN)


context   MakeRental:: availability() :  Boolean
post:
          let todayAvailability: Integer= self.pickUpBranch.
            groupAvailability->select(gA|gA.carGroup=
      self.carG).quantity
          let validRental:RentalAgreement=Reservation.allInstances()
            ->reject(r|r.oclIsTypeOf(CanceledReservation))->
            select(r|r.carGroup=self.carGroup)-
>forAll(r|r.car=isEmpty)->
            union(RentalAgreement.allInstances()->select(r.car->
            isNotEmpty() and r.isGroup=self.carG)
          let sumReservation=validReservation->
              select(r|not(r.pickUpBranch=self.pickUpId)
              and r.dropOffBranch=self.pickUpId and
r.agreedEnding.date()
               < self.beginning and not(r.oclIsTypeOf(ClosedRental))
          let decReservation=validReservation->select(r|r.beginning >
             now() and r.pickUpBranch=self.pickUpId and
            (not(r.dropOffBranch=self.pickUpId) or
             r.agreedEnding.date() < self.date))
           in
            result=(todayAvailability+sumReservation) > decReservation


context   MakeRental:: pickUpBranch() :  Branch
post:
          let branches:Set(Branch) = Branch.allInstances()->
            select(name=self.pickUpId)
          in
          branches->size()=1 implies result=branches->any()


context   MakeRental:: dropOffBranch() : Branch
post:
          let branches:Set(Branch) = Branch.allInstances()->
            select(name=self.dropOffId)
```

```
          in
          branches->size()=1 implies result=branches->any()


context   MakeRental:: apply()
post:
          let getCountries:Set(Country) =self.countries-> forAll(name
|
             Country.allInstances()->select(c|c.name=name))
          in
          renter.oclIsTypeOf(Customer) and rental.oclIsNew() and
          rental.oclIsTypeOf(RentalAgreement) and
          rental.beginning=beginning  and rental.renter=renter and
          rental.initEnding=self.ending and
          rental.pickUpBranch=self.pickUpBranch and
          rental.dropOffBranch=self.dropOffBranch and
          rental.country= getCountries and answerRental=r
```

MakeWalkInRental

```
context   MakeWalkInRental:: candidateCars() : CarGroup
post:
          let carsAv: Set(Car)= self.pickUpBranch.carsAvailableNow
          let carsFreed: Set(Car)= self.pickUpBranch.car->
            select(c|c.assigned? and c.rentalAgreement->
            exists(r|r.beginning.date()=today() and
             not(r.oclIsTypeOf(OpenedRental)) and
            not(r.oclIsTypeOf(GuaranteedRental)) and now –beginning
            >(minute(90))))
          result= carsAv->union(self.carsFreed)->select(c|c.carGroup=
          self.carG)


context   MakeWalkInRental:: apply()
post:
          self.oclAsType(MakeRental).^apply and self.beginning >
          self.time and self.beginning.day()=self.time.day()
          and   self.answerRental.car=
            candidateCars->sortedBy(mileageFromLastService)->first()
```

MakeWalkInRentalWithCarModel

```
context   MakeWalkInRentalWithCarModel:: candidateCars() : CarModel
post:
          let carsAv: Set(Car)= self.pickUpBranch.carsAvailableNow
          in
          let carsModAv: Set(Car)= carsAv-
>select(c|c.carModel=self.carM)
          in
          result=
            if carsModAv->notEmpty() then
             carsModAv
          else
            carsAv-> select(c|c.carGroup= self.carG)
          end if
```

MakeReservation

```
context   MakeReservation:: apply()
post:
          self.oclAsType(MakeRental).^apply() and
          self.answerRental.reservationDate=self.time and
          self.answerRental.carGroup=carG
```

MakeReservationWithCarModel

```
context   MakeReservationWithCarModel:: apply()
post:
          self.oclAsType(MakeReservation).^apply() and
          self.answerRental.carModel=self.carM
```

ExistingPersonCheck

```
context   ExistingPersonCheck:: apply()
post:
          let pers: Set(Person)=   EU_RentPerson.allInstances()->
            select(p|p.id=self.id)
          in
          self.answerExists=pers->isNotEmpty() and pers->isNotEmpty()
            implies self.answerPerson=pers->any()
```

CancelCurrentRental

```
context   CancelCurrentRental:: apply()
post:
          RentalAgreement.allInstances()->excludes(self.rental)
```

GuaranteeReservation

```
context   GuaranteeReservation:: apply()
post:
          self.reservation.oclIsTypeOf(GuaranteedReservation)
          and self.reservation.oclAsType(GuaranteedReservation).
          creditCardNumber=self.creditCard
```

EndWalkInRental

```
context   EndWalkInRental:: apply
post:
```

GetReservation

```
context   GetReservation:: reservationExists() :     Boolean
post:
          result=self.rental.oclIsKindOf(Reservation) and
          not(self.rental.oclIsKindOf(CanceledReservation))


context   GetReservation:: apply()
post:
```

```
          answerReservation= self.person.Rental
          Agreement->select(r|  r.beginning=self.beginning)
```

RentalExtension

```
context   RentalExtension:: openedRentalExists() :   Boolean
post:
          result=self.rental.oclIsKindOf(OpenedRental) and
          not(self.rental.oclIsKindOf(CanceledReservation))


context   RentalExtension:: maintenanceNotNeeded() : Boolean
post:
          result=not(self.rental.car.oclIsTypeOf(NeedsMaintenance))


context   RentalExtension:: apply()
post:
          let newExtension:Extension =
            self.openedRental.newEndings.last().extension
          in
          self.rental.oclIsTypeOf(ExtendedRental) and
          self.newExtension.oclIsNew() and self.newExtension.extension
          Done=self.time and self.newExtension.dateTime=self.newEnd
```

CancelReservation

```
context   CancelResevation:: apply()
post:
          self.reservation.oclIsTypeOf(CanceledCustomer)
```

CancelGuaranteedReservation

```
context   CancelGuaranteedReservation:: allInstances() : Boolean
post:
          CancelReservation.allInstances()->select(cr
          |cr.reservation.oclIsKindOf(GuaranteedReservation) and
          cr.reservation.beginning.date()=today())


context   CancelGuaranteedReservation:: apply()
post:
          self.oclAsType(CancelReservation).^apply() and
          charge(self.reservation.oclAsType(GuaranteedCanceled).fine,
          self.oclAsType(GuaranteedReservation).creditCardNumber)
```

CheckTodayReservationWithoutCar

```
context   CheckTodayReservationWithoutCar:: allInstances() : Boolean
post:
          result=CheckTodayResWithoutCarBasic.allInstances()->
            union(CheckTodayResWithoutCarDerived.allInstances())
```

```
context   CheckTodayReservationWithoutCar:: apply()
post:
          let pendantReservation: Set(Reservation)=
          Reservation.allInstances-
>select(r|r.beginning.date()=today() and
            r.car->isEmpty)-> select(r|r.pickUpBranch=
            self.reservation.pickUpBranch)
          let carG: CarGroup=self.reservation.car.carGroup
          let pendGroupR:Set(Reservation)=self.pendantReservation
          ->select(r|r.carGroup=self.carG or
r.carGroup=self.carG.better or
            r.carGroup=self.carG.worse)
          in
          if self.reservation.beginning.date()=today and
            self.pendGroup->isNotEmpty() then
             self.pendGroup->any().car=self.reservation.car
          end if
```

CUSTOMER MANAGEMENT EVENTS

RecordCustomer

```
context   RecordCustomer:: apply()
post:
          p.oclIsNew() and and p.oclIsTypeOf(EU_RentPerson) and
          p.id=self.id and p.name=self.name and p.address=self.address
          and p.birthdate= self.birthdate and answerPerson=p
```

RecordDriverData

```
context   RecordDriverData:: apply()
post:
          dl.oclIsNew() and dl.oclIsTypeOf(DrivingLicense) and
          dl.number=self.drivingLicenseNumber and
          dl.issue=self.issue and dl.expiration=self.expiration
          and dl.EU_RentPerson = self. person
```

JoinLoyaltyIncentiveScheme

```
context   JoinLoyaltyIncentiveScheme:: apply() : Branch
post:
          self.person.oclIsTypeOf(LoyaltyMember) and
          self.person.oclAsType(LoyaltyMember).membershipDate=today()
```

CancelLoyaltyMembership

```
context   CancelLoyaltyMembership:: isMember() :     Boolean
post:
          self.person.oclIsTypeOf(LoyaltyMember)


context   CancelLoyaltyMembership:: apply() : Boolean
post:
          not(self.person.oclIsTypeOf(LoyaltyMember))
```

DefaultingCustomer

```
context   DefaultingCustomer:: apply() : Boolean
post:
          let fault:FaultSeriousness= self.rental.faultSeriousnes->
           select(f|f.badExperience.type=paymentProblem)
          in
          self.fault.oclIsNew and self.fault.degree= self.problem
          Seriousness  and self.answerToBeBlacklisted=
            blacklistingCriteriaAchieved(self.person)
          and not self.person.oclIsTypeOf(LoyaltyMember)
```

CancelCustomersReservations

```
context   CancelCustomersReservations:: apply()
post:
          let openRes: Reservation= self.person.rentalsAsRenter@pre->
           select(r| r.oclIsTypeOf(Reservation))->
            reject(r.oclIsTypeOf(OpenedRental) or
            r.oclIsTypeOf(CanceledReservation))
          in
          openRes->forAll(r| r.oclIsTypeOf(CanceledReservation)
          and r.oclAsType(CanceledReservation).motivation=
self.motivation)
```

BlacklistCustomer

```
context   BlacklistCustomer:: apply()
post:
          self.person.oclIsTypeOf(Blacklisted) and
          self.person.oclAsType(Blacklisted).blacklistedDate=today()
and
          self.oclAsType(CancelCustomersReservations).^apply()
```

GetLoyaltyCandidates

```
context   GetLoyaltyCandidates:: apply()
post:
          let curBranch:Branch= Branch.allInstances()->
          select(b|b.name=currentBranchName())
          let closedRentalsLastYear(p:EU_RentPerson)=
p.RentalAgreement->
            select(oclIsTypeOf(ClosedRental))->select(cR| Now() –
            cR.beginning < Year(1))
          in
          answerCustomers= curBranch.branchCustomer->reject(
            oclIsTypeOf(LoyaltyMember) or oclIsTypeOf(Blacklisted))->
            select(p| p.faults->isEmpty() and
      self.closedRentalsLastYear(p)
            ->count>=4 and self.closedRentalsLastYear(p).actualReturn->
            collect(d|d.date())->includes(today()))
```

GetTodayBlacklisted

```
context   GetTodayBlacklisted:: apply()
post:
          answerCustomers= Blacklisted.all
```

```
Instances()->select(b|b.blacklistedDate=today())
```

PRICING AND DISCOUNTING MANAGEMENT EVENTS

OfferSpecialAdvantatges

```
context   OfferSpecialAdvantatges:: apply()
post:
          let basePr:Money=self.reservation.basicPrice
          let bestPrice:Money=self.reservation.bestPrice
          let reservationTimeDiscountPerDuration(rd: RentalDuration)
          =self.reservation.applicableDiscountPerDuration
            ->select(d|d.reservationTime)
          let bestRentalDiscountPerDuration(rd:RentalDuration,
          basicPrice: Money) : Discount=
            self.rentalApplicableDiscountPerDuration(rd)->
            reject(disAct: Discount|
            self.rentalApplicableDiscountPerDuration(rd)->
            exists(disOther:Discount| apply(disOther,
            rd).isBetter(apply(disAct, rd)))->any()
          let bestSpD: Money = self.bestDurationPrices->iterate(elem;
            tup : Tuple {accInterval: Duration=self.onRentInterval,
            accPrice:   Money=0} |
            let timeMax:Duration= durationT(elem.timeUnit,
             elem.maximumDuration)
            let timeMin:Duration= durationT(elem.timeUnit,
             elem.minimumDuration)
            let numInt:Integer =
              if tup.accInterval >= timeMax then
                 tup.accInterval/timeMax
              else
                 tup.accInterval/timeMin
            in
            Tuple {accInterval:Duration=
            (if tup.accInterval >= timeMax then
                   tup.accInterval%timeMax
               else
                    tup.accInterval%timeMin
               endif),
              accPrice:Money= tup.accPrice+
              numInt*apply(self.bestRentalDiscountPerDuration
              (elem.rentalDuration, elem.price),
                  elem.rentalDuration)}).accPrice
          in
          answerSOptions=Sequence{}->append(Tuple{id="Base price",
            desc= basePr.toString})->
          append(Tuple{id="Best price", desc=bestPr.toString}) ->
            append(Tuple{desc="Special Advantatges", desc=bestSpD})
```

OfferPaymentWithPoints

```
context   OfferPaymentWithPoints:: apply()
post:
          let basePr:Money=self.reservation.basicPrice
          let points:Integer=points(basePr)
          in
          self.oclAsType(OfferSpecialAdvantatges).^apply() and
          if (self.reservation.renter.oclIsTypeOf(LoyaltyMember)
          and self.oclAsType(LoyaltyMember).availablePoints>=points
and
          (self.reservation.beginning.day()-
            self.reservation.reservationDate.day()) >=day(14))
              self.answerOptions=self.answerSOptions->
             append(Tuple(id="Points", desc=points)
          else
             self.answerOptions=self.answerSOptions
          end if
documentation:
          Accuracy is difficult to define in this operation,
          because of a non automatic definition of
          discounts...etc
```

ChooseDiscountOption

```
context   ChooseDiscountOption:: validOption() :     Boolean
post:
          result= (self.selectedOption="Special Advantatges") or
           (self.selectedOption="Best Price") or
(self.selectedOption=
             "Points")


context   ChooseDiscountOption:: apply() :  Boolean
post:
          if (selectedOption="Special Advantatges") then

self.reservation.oclIsTypeOf(ReservationWithSpecialDiscount)
          else
             if (selectedOption=" Points") then

self.reservation.oclIsTypeOf(PointsPaymentReservation)
            end if
          end if
```

ShowBestBasePrice

```
context   ShowBestBasePrice:: apply() :  Boolean
post:
          self.answerPrice=self.reservation.bestPrice.toString()
```

ShowBestPrice

```
context   ShowBestPrice:: apply() :  Boolean
post:
          self.answerPrice=self.reservation.basicPrice
```

RecordNewDiscount

```
context   RecordNewDiscount:: durationsExist() :     Boolean
post:
          self.applicableDurations->forAll(dur |
            RentalDuration.allInstances() ->exists(d|d.name=dur))


context   RecordNewDiscount:: groupsExist() :  Boolean
post:
          self.applicableGroups->forAll(group |
CarGroup.allInstances()
             ->exists(g|g.name=group))


context   RecordNewDiscount:: apply() :  Boolean
post:
          let getDurations:Set(RentalDuration)=
self.applicableDurations->
            forAll(name|RentalDuration.allInstances()-
      >select(r|r.name=name)
          let getGroups:Set(CarGroup) = self.applicableGroups->
            forAll(name| CarGroup.allInstances()-
      >select(cg|cg.name=name)
          in
            dis.oclIsNew() and dis.oclIsTypeOf(Discount) and dis.name=
            self.name and dis.effect=self.effect and dis.description=
            self.description and
      dis.reservationTime=self.reservationTime
            and dis.beginningDate=self.time.date() and dis.carGroup=
            self.getGroups and dis.rentalDuration=self.getDurations
```

CloseDiscount

```
context   CloseDiscount:: apply() :  Boolean
post:
          self.discount.oclIsTypeOf(ClosedDiscount) and
          self.discount.oclAsType(ClosedDiscount).endingDate
            =self.time.date()
```

RecordNewRentalDuration

```
context   RecordNewRentalDuration:: previousExists() :     Boolean
post:
          previousName->notEmpty() implies
RentalDuration.allInstances()
            ->exists(rd| rd.name= self.durationName)


context   RecordNewRentalDuration:: apply() :  Boolean
post:
          let previousRC:RentalDuration= RentalDuration.allInstances()
            ->select(r|r.name=self.previousName)
          in
          rc.oclIsNew() and rc.oclIsTypeOf(RentalDuration) and
      rc.name=
           self.name and rc.minimumDuration=self.minimumDuration and
          rc.maximumDuration= self.maximumDuration and
          rc.timeUnit=self.timeUnit and
```

```
          if previousRC->notEmpty() then
            rc.shorter=self.previousRC->any() and
          rc.longer=self.previousRC->any().longer@pre
          else
            rc.shorter->isEmpty() and rc.longer=RentalCategory.
            allInstances() ->any(r|r.shorter@pre->isEmpty)
          end if
```

NewCarGroupDurationPrice

```
context   NewCarGroupDurationPrice:: apply() :  CarGroup
post:
          cgdp.oclIsNew() and cgdp.oclIsTypeOf(CarGroupDurationPrice)
and
            cgdp.price=self.price and cgdp.carGroup=self.carG and
            cgdp.rentalDuration=duration
```

NewCGDPForNewDuration

```
context   NewCGDPForNewDuration:: apply()
post:
          cgdp.oclIsNew() and cgdp.oclIsTypeOf(CarGroupDurationPrice)
and
            cgdp.price=self.price and cgdp.carGroup=self.carG and
            cgdp.rentalDuration=duration
```

ChangeCarGroupDurationPrice

```
context   ChangeCarGroupDurationPrice:: apply()
post:
          CarGroupDurationPrice.allInstances()->select(cgdp|
          cgdp.carGroup=self.carG and
          cgdp.rentalDuration=duration).price=self.price
```

CAR ALLOCATION EVENTS

CarAllocationWithAnExceptionRule

```
context   CarAllocationWithAnExceptionRule:: apply()
post:
          if upWalkInPossible then
            answerOptions->includes("walk-in")
          end if
          if 2upgradePossible then
           answerOptions->includes("bumped-upgrade")
          end if
          if downgradePossible then
           answerOptions->includes("downgrade")
          end if

          if transferPossible then
           answerOptions->include("transfer")
          end if
          if servicePossible then
           answerOptions->include("service")
          end if
```

CarAllocationAutomatic

```
context   CarAllocationAutomatic:: apply()
post:
          let modelAvail(m:CarModel): ModelAvailability=
self.curBranch.
            modelAvailability@pre-> select (mA| mA.carModel=m)
          in
          if (groupAvail(self.curGroup)->isEmpty() or (groupAvail
            (self.curGroup).quantity@pre<(self.demXGroup->select
            (self.curGroup).demand@pre)) and self.upgradePossible then
          -- Do upgrade
          self.reservation.car->isNotEmpty() and self.curBranch.
            carsAvailable@pre->select(c|c.carGroup=
          self.upgradeGroup)->includes(self.reservation.car)
          else
            if self.curModel->isNotEmpty() and
             self.modelAvail(self.curModel) then
            -- Model desired
            self.reservation.car->isNotEmpty() and self.curBranch.
            carsAvailable@pre->select(c|c.carModel=
            self.curModel)->includes(self.reservation.car)
            else
              -- Model with lower demand
               if self.availGroup(curGroup)->isNotEmpty then
                self.reservation.car->isNotEmpty() and
                self.reservation.car.carModel=
self.curGroup.carModels->
                 sortedBy(cM|self.availModel(cM)@pre-self.demXModel->
                select(d|d.carModel=cM).demand@pre)->first()
                end if
            end if
          end if
```

CarAllocationKind

```
context   CarAllocationKind:: apply()
post:
          if self.upgradePossible or self.groupAvail(self.curGroup)
then
            answerKind=Automatic
          else
            if upWalkInPossible or 2upgradePossible or
downgradePossible or
             transferPossible or servicePossible then
           answerKind=Exception
           else
           answerKind=InExtremis
           end if
          enf if
```

CarAllocationExceptionOption

```
context   CarAllocationExceptionOption:: validOption() :   Boolean
post:
          result=(option="walk-in" and upWalkInPossible)  or
          (option="bumped-upgrade" and 2upgradePossible)  or
          (option= "downgrade" and downgradePossible) or
          (option="transfer" and transferPossible) or
```

```
          (option="service" and servicePossible)


context   CarAllocationExceptionOption:: apply()
post:
          let curGroup:CarGroup=self.reservation.carGroup
          let upgradeGroup:CarGroup=self.reservation.carGroup.better
          let  downgradeGroup:CarGroup=self.reservation.carGroup.worse
          let 2upgradeGroup:CarGroup=
           if upgradeGroup->isEmpty then {}
           else upgradeGroup.better
          let curBranch: Branch= self.reservation.pickUpBranch
          in
          if option="walk-in" then
            self.reservation.car->isNotEmpty() and self.curBranch.
            availableCars@pre->select(c|carGroup=self.upgradeGroup)->
            includes(self.reservation.car)
          end if
          if option="bumped-upgrade" then
            self.reservation.car->isNotEmpty() and
          self.reservation.car.carGroup=self.upgradeGroup and
           let intermediateR: Reservation=self.curBranch.nextDayR->
             select(r|r.car@pre= self.reservation.car)
           in
           self.intermediateR.car.carGroup=self.2upgradeGroup
          and
           not(self.intermediateR.renter.oclIsKindOf(Loyalty
          Member))   implies self.curBranch.nextDayR-
>select(r|r.carGroup=
          upgradeGroup and r.renter.oclIsKindOf(LoyaltyMemeber))->
            forAll(r|r.car.cargroup=2upgradeGroup)
          end if
          if option= "downgrade" then
           self.reservation.car->isNotEmpty() and
          self.reservation.car.carGroup= self.downgradeGroup
          end if
          if option="transfer" then
          self.reservation.car->isNotEmpty() and
          self.curBranch.transferAgreement[transferor]->select(
          tA.transferor.GroupAvailability->select(gA|ga.carGroup=
          self.curGroup).quantity> tA.transferor.demandXGroup->
          select(d|d.carGroup=self.curGroup).demand)->sortedBy
          (tA|tA.expectedTime < self.reservation.beginning-now()-
          preparingTime())->first().car-
      >select(c|c.carGroup=self.curGroup)
          ->includes(self.reservation.car)
          end if

          if option="service" then
           self.reservation.car->isNotEmpty() and
          self.reservation.car.oclIsKindOf(maintenance
          Scheduled)@pre  and self.reservation.car.ocllAsType
          (MaintenanceScheduled@pre.beginningDate <>tomorrow())
          and not(self.reservation.car.oclIsKindOf
(maintenanceScheduled))
            and self.reservation.car.carGroup=self.reservation.carGroup
          end if
```

CarAllocationExtremisOption

```
context   CarAllocationExtremisOption:: validOption() :    Boolean
post:
          result=( option="delay" or  option="competitor")


context   CarAllocationExtremisOption:: apply()
post:
          if option="delay" then
            self.reservation.car->isEmpty()
          end if
          if option="competitor" then
             not(self.reservation.car.oclIsTypeOf(OwnCar))
          end if
```

CarAllocationWithAnExtremisRule

```
context   CarAllocationWithAnExtremisRule:: apply()
post:
          self.answerOptions.includes("delay") and
            self.answerOptions.includes("competitor")
```

CreateTransferAgreement

```
context   CreateTransferAgreement:: notPreviousAgreement() :
      Boolean
post:
          result=self.receiverBranch.transferor->excludes(self.branch)


context   CreateTransferAgreement:: apply()
post:
          let reverseTransAg:Set(TransferAgreement) =
TransferAgreement.
            allInstances()->select(ta|ta.receiver=self.branch and
          ta.transferor=self.receiverBranch)
          in
          ta.oclIsNew() and ta.oclIsTypeOf(TransferAgreement)
          and ta.transferor=self.branch and
ta.receiver=self.receiverBranch
             and answerTransAg=ta
          and
          if reverseTransAg->notEmpty then
            answerDataNeeded=false and
      ta.distance=reverseTransAg.distance
             and ta.expectedTime=reverseTransAg.expectedTime
          else
           answerDataNeeded=true
          end if
```

CancelTransferAgreement

```
context   CancelTransferAgreement:: transferAgExists() :   Boolean
post:
          result=TransferAgreement.allInstances()->exists(ta|
          ta.transferor=self.branch and
ta.receiver=self.receiverBranch)
```

```
context   CancelTransferAgreement:: apply()
post:
          TransferAgreement.allInstances()->excludes(ta|
          ta.transferor=self.branch and
ta.receiver=self.receiverBranch)
```

ChangeTransferAgreementData

```
context   ChangeTransferAgreementData:: apply()
post:
          self.transferAg.distance(km)=self.distance and
          self.transferAg.expectedTime(h)=self.expectedTime
```

IntroduceTransferData

```
context   IntroduceTransferData:: apply()
post:
          self.transAg.distance=self.distance and
            self.transAg.expectedTime= self.expectedTime
```

GetCarsToBeTransferred

```
context   GetCarsToBeTransferred:: apply()
post:
          let carsToBeTrans:Set(Car)= self.branch.car-
>select(c|c.rental
          Agreement->exists(r|r.beginning=tomorrow() and
          not(r.oclIsKindOf(CanceledReservation) and
            r.pickUpBranch<>self.branch)
          in
          self.answerTransfers=carsToBeTrans->forAll(c| Tuple(car
          RegN=c.registrationNumber,  destination= c.rentalAgreement->
          select(r|r.beginning=tomorrow()).pickUpBranch.name))
```

TransferOwnership

```
context   TransferOwnership:: validCar() :     Boolean
post:
          result=self.car.rentalAgreement->exists(r| not(r.oclIsKindOf
            (CanceledReservation)) and not(r.oclIsKindOf(OpenedRental))
      and
             r.pickUpBranch=self.branch))


context   TransferOwnership:: validBranch() :  Boolean
post:
          result = self.car.Branch<>self.branch


context   TransferOwnership:: apply()
post:
          self.car.branch@pre<>self.car.branch and
             self.car.branch=self.branch
```

SellCarsInNeed

```
context   SellCarsInNeed:: apply()
post:
          self.branch.carsAvailable@pre->select(c|
            (today()-c.acquisitionDate>=year(1)) or
            (c.currentMileage>40.000))-
      >forAll(c|c.oclIsTypeOf(ToBeSoldCar))
```

CancelNoShowReservations

```
context   CancelNoShowReservations:: apply()
post:
          Reservation.all
          Instances->select(r|r.beginning=today())->forall(
            r.oclIsTypeOf(CanceledCompany) and r.oclAsType(
            CanceledCompany).motivation= CancellingMoTivation::no-show)
```

CarAllocationDefinitions

```
context   CarAllocationDefinitions:: curBranch() :   Branch
post:
          result= self.reservation.pickUpBranch


context   CarAllocationDefinitions:: curGroup() :    CarGroup
post:
          result=self.reservation.carGroup


context   CarAllocationDefinitions:: upgradeGroup() :      CarGroup
post:
          result=self.reservation.carGroup.better


context   CarAllocationDefinitions:: groupQuota(b : Branch) :
      Integer
post:
          result= curGroup.carGroupQuota-
>select(branch=curBranch).quota


context   CarAllocationDefinitions:: groupAvail(g : CarGroup) :
      GroupAvailability
post:
          result= self.curBranch.groupAvailability@pre-> select
          (gA| gA.carGroup=g)


context   CarAllocationDefinitions:: upgradePossible() :   Boolean
post:
          result=self.groupAvail(self.upgradeGroup)->isNotEmpty()
          and self.groupAvail(self.upgradeGroup).quantity@pre -
          self.demXGroup-
>select(d|d.carGroup=self.upgradeGroup).demand@pre
            >0.1*groupQuota(self.curBranch)


context   CarAllocationDefinitions:: demXModel() :   DemandXModel
```

```
post:
          result=self.reservation.pickUpBranch.demandXModel


context   CarAllocationDefinitions:: demXGroup() :   DemandXGroup
post:
          result=self.reservation.pickUpBranch.demandXGroup
```

ExtendedCarAllocationDefinitions

```
context   ExtendedCarAllocationDefinitions:: downgradeGroup() :
      CarGroup
post:
          result= self.reservation.carGroup.worse


context   ExtendedCarAllocationDefinitions:: 2upgradeGroup() :
      CarGroup
post:
          result=
           if upgradeGroup->isEmpty then {}
           else upgradeGroup.better


context   ExtendedCarAllocationDefinitions:: upWalkInPossible():
      Boolean
post:
          result= if self.upgradeGroup->isNotEmpty() then
            self.groupAvail(self.upgradeGroup)->isNotEmpty and
            self.groupAvail(self.upgradeGroup).quantity@pre -
            self.demXGroup->select
            (d|d.carGroup=self.upgradeGroup).demand@pre  >0
          else
           False


context   ExtendedCarAllocationDefinitions:: 2upgradePossible():
      Boolean
post:
          result=
            if self.2upgradeGroup->isNotEmpty() then
             (self.curBranch.nextDayR.car->collect(carGroup)->includes
             (upgradeGroup) or self.groupAvail(self.upgradeGroup)) and
             self.groupAvail(self.2upgradeGroup)->isNotEmpty and
             self.groupAvail(self.2upgradeGroup).quantity@pre -
             self.demXGroup->select(d|d.carGroup=self.2upgradeGroup).
             demand@pre  >0.1*self.groupQuota(self.curBranch,
             self.2upgradeGroup)
          else
           False


context   ExtendedCarAllocationDefinitions::
downgradePossible():Boolean
post:
          result=
           if self.downgradeGroup->isNotEmpty() then
            self.groupAvail(self.downgradeGroup)->isNotEmpty and
            self.groupAvail(self.downgradeGroup).quantity@pre -
            self.demXGroup->select(d|d.carGroup=self.downgradeGroup).
```

```
            demand@pre  >0.1*self.groupQuota(self.curBranch,
            self.downgradeGroup)
          else
           False


context   ExtendedCarAllocationDefinitions:: transferPossible():
      Boolean
post:
          result=self.curBranch.transferAgreement[transferor]->exists
            (tA|tA.expectedTime < self.reservation.beginning-now()-
            preparingTime() and tA.transferor.GroupAvailability-
      >select(
            gA|ga.carGroup=self.curGroup).quantity >
            tA.transferor.demandXGroup-> select(d|d.carGroup
            =self.curGroup).demand )


context   ExtendedCarAllocationDefinitions:: servicePossible() :
Boolean
post:
          result= self.curBranch.car-> exists(c|
            c.oclIsKindOf(MaintenanceScheduled) and
             c.oclAsType(MaintenanceScheduled).beginningDate
      <>tomorrow())


context   ExtendedCarAllocationDefinitions:: downgradeGroup() :
      CarGroup
post:
          result= self.reservation.carGroup.worse


context   ExtendedCarAllocationDefinitions:: 2upgradeGroup() :
      CarGroup
post:
          result=
           if upgradeGroup->isEmpty then {}
           else upgradeGroup.better


context   ExtendedCarAllocationDefinitions:: upWalkInPossible():
      Boolean
post:
          result= if self.upgradeGroup->isNotEmpty() then
            self.groupAvail(self.upgradeGroup)->isNotEmpty and
            self.groupAvail(self.upgradeGroup).quantity@pre -
            self.demXGroup->
            select(d|d.carGroup=self.upgradeGroup).demand@pre  >0
          else
           False


context   ExtendedCarAllocationDefinitions:: 2upgradePossible():
      Boolean
post:
          result=
          if self.2upgradeGroup->isNotEmpty() then
           (self.curBranch.nextDayR.car->collect(carGroup)->includes
            (upgradeGroup) or self.groupAvail(self.upgradeGroup)) and
            self.groupAvail(self.2upgradeGroup)->isNotEmpty and
```

```
            self.groupAvail(self.2upgradeGroup).quantity@pre -
            self.demXGroup->select(d|d.carGroup=self.2upgradeGroup).
            demand@pre  >0.1*self.groupQuota(self.curBranch,
            self.2upgradeGroup)
          else
           False


context   ExtendedCarAllocationDefinitions::
downgradePossible():Boolean
post:
          result=
           if self.downgradeGroup->isNotEmpty() then
            self.groupAvail(self.downgradeGroup)->isNotEmpty and
            self.groupAvail(self.downgradeGroup).quantity@pre -
            self.demXGroup->select(d|d.carGroup=self.downgradeGroup).
            demand@pre  >0.1*self.groupQuota(self.curBranch,
            self.downgradeGroup)
          else
           False


context   ExtendedCarAllocationDefinitions::
transferPossible():Boolean
post:
          result=self.curBranch.transferAgreement[transferor]->
            exists(tA|tA.expectedTime < self.reservation.beginning-
      now()-
            preparingTime() and tA.transferor.GroupAvailability-
      >select(
            gA|ga.carGroup=self.curGroup).quantity>
            tA.transferor.demandXGroup ->
            select(d|d.carGroup=self.curGroup).demand )


context   ExtendedCarAllocationDefinitions:: servicePossible() :
      Boolean
post:
          result= self.curBranch.car-> exists(c|c.oclIsKindOf
            (maintenanceScheduled) and
            c.oclAsType(MaintenanceScheduled). beginningDate
            <>tomorrow())
```

AllocationEstimators

```
context   AllocationEstimators:: apply()
post:
          let nextDayR:Set(Reservation)= self.branch.nextDayR
          in
          self.answerLoyaltyReservation=self.nextDayR->
            select(r|r.renter.oclIsKindOf(LoyaltyMember))
          ->sortedBy(reservationDate) and
          self.answerGuaranteeReservation= self.nextDayR->
            select(r|r.oclIsKindOf(GuaranteedRental))->
            reject(r|self.answerLoyaltyReservation->includes(r))->
            sortedBy(reservationDate) and
          self.answerOtherReservation= self.nextDayR->reject(r|
          self.answerLoyaltyReservation->includes(r) or
          self.answerGuaranteeReservation->includes(r))->
            sortedBy(reservationDate)
```

CAR PREPARATION AND MAINTENANCE EVENTS

GetCarsToBePrepared

```
context   GetCarsToBePrepared:: apply() :  Set(Reservation)
post:
          answerCar=self.branch.car->select(c|c.rentalAgreement->
            exists(r| .beginning.date()=today())->sortedBy (c|c.rental
          Agreement.beginning->select(d|d.date()=today()))
```

CarPrepared

```
context   CarPrepared:: apply()
post:
          AssignedCar.allInstances->select(ac|ac.car=self.car
          and
c.rentalAgreement.beginning=today()).oclIsKindOf(Prepared)
```

RecordNewMileage

```
context   RecordNewMileage:: validMileage() :  Boolean
post:
          self.newMileage > self.car.currentMileage@pre


context   RecordNewMileage:: apply() :  Boolean
post:
          self.car.oclAsType(OwnCar).currentMileage=newMileage
          and
self.answerSellCar=(not(self.car.oclIsKindOf(NeedMaintencance)
          and not(damagesDetected?) and
          self.car.oclIsKindOf(NeedToBeSoldCar))and
          if self.car.oclIsKindOf(NeedMaintenance) then
            self.car.oclIsKindOf(MaintenanceScheduled) and
          self.car.oclAsType(MaintenanceScheduled).beginningDate=
            getMaintenanceDate()
          end if
```

DamagesEvaluation

```
context   DamagesEvaluation:: apply() :  CarGroup
post:
          self.answerOwnCar?=self.car.oclIsKindOf(OwnCar) and
          self.answerCar=self.car
```

RecordDamages

```
context   RecordDamages:: rentalIsClosed() :  CarGroup
post:
          result=self.rental.oclIsKindOf(ClosedRental)


context   RecordDamages:: apply() :  CarGroup
```

```
post:
          let carDam:BadExperience= CarDamages.allInstances()->any()
          let closedR:closedRental=
self.rental.oclAsType(closedRental)
          in
          self.answerCar=self.rental.car and
          -- cost to the renter
          self.closedR.badExp->includes(carDam) and self.closedR.
            faultSeriousness->select(fs|fs.badExp=carDam).degree=
            self.damageDegree and self.closedR.damageCost=self.cost and
            charge(self.closedR.creditCarNumberDamages, self.cost) and
          -- schedule reparations
          self.rental.car.oclIsKindOf(RepairsScheduled) and
          self.rental.car.oclAsType(RepairsScheduled).beginningDate=
            scheduleReparations(self.rental) and
          self.answerToBeBlacklisted=blacklistingCriteriaAchieved
            (self.rental.renter) and not
          self.rental.driver.oclIsTypeOf(LoyaltyMember)
```

ScheduleMaintenance

```
context   ScheduleMaintenance:: carNeedsMaintenance() :    Boolean
post:
          result=self.car.oclIsTypeOf(NeedsMaintenance)


context   ScheduleMaintenance:: apply() :  Boolean
post:
          self.car.oclIsTypeOf(MaintenanceScheduled) and
          self.car.oclAsType(MaintenanceScheduled).beginningDate=
beginning
```

EndOfMaintenance

```
context   EndOfMaintenance:: carWasBeingMaintained() :     Boolean
post:
          result=self.car.oclIsTypeOf(MaintenanceScheduled) and
          self.car.oclAsType(MaintenanceScheduled).beginningDate<
now())


context   EndOfMaintenance:: apply() :  Boolean
post:
          self.car.mileageFromLastService= self.car.currentMileage and
            self.car.lastMaintenanceDate= today() and
          not(self.car.oclIsKindOf(NeedMaintenance)) and

self.answerSellCar=(not(self.car.oclIsKindOf(RepairsScheduled)
            and self.car.oclIsKindOf(NeedToBeSold))
```

EndOfRepairs

```
context   EndOfRepairs:: carWasBeingRepaired() :  Boolean
post:
          result=self.car.oclIsTypeOf(RepairsScheduled) and
          self.car.oclAsType(RepairsScheduled).beginningDate< now())
```

```
context   EndOfRepairs:: apply() :  Boolean
post:
          not(self.car.oclIsKindOf(RepairsScheduled)) and

self.answerSellCar=(not(self.car.oclIsKindOf(MaintenanceScheduled)
            and self.car.oclIsKindOf(NeedToBeSold))
```

CAR PICK-UP AND RETURN EVENTS

ExistingReservationForToday

```
context   ExistingReservationForToday:: reservationExists() :
      Boolean
post:
          Reservation.allInstances->exists(r|r.beginning=today() and
            r.renter=self.person and
      r.oclIsNotKindOf(CanceledReservation))


context   ExistingReservationForToday:: apply() :  Boolean
post:
          let res:Reservation= Reservation.allInstances()->
            select(r|r.beginning=today() and r.renter=self.person)
          in
          self.answerReservation=res and self.answerExpectLatePrep?=
            not(self.answerReservation.assignedCar->notEmpty() and
            (self.answerReservation.assignedCar.expectedPreparedTime<=
            self.rental.beginning) and
            self.answerReservation.assignedCar.
            oclIsKindOf(Prepared) implies self.answerReservation.
            assignedCar.oclAsType(Prepared).actualTime<=
            self.rental.beginning)  ) ))
```

AddDriverToRental

```
context   AddDriverToRental:: apply() :  carGroup
post:
          self.rental.driver->includes(self.driver)
```

OpenRental

```
context   OpenRental:: apply() :  Money
post:
          self.rental.oclIsKindOf(OpenedRental) and
          self.rental.oclAsType(OpenedRental).actualPick-UpTime=now()
and
            not(self.rental.oclIsKindOf(ClosedRental)) and
            not(self.rental.oclIsKindOf(ExtendedRental)) and
          self.answerLatePreparation= self.rental.AssignedCar.
            oclAsType(Prepared).actualTime-self.rental.beginning
      >hour(1)
```

ApologiseForLatePreparation

```
context   ApologiseForLatePreparation:: apply() :  Boolean
post:
```

```
          sendApologiseLetter(self.reservation.renter)
```

ApologisePlusReimbursement

```
context   ApologisePlusReimbursement:: apply() :  Boolean
post:
          let hourlyPaid: Money= self.reservation.bestDurationPrices->
            select(b|b.rentalDuration.minimumDuration=1
          and b.rentalDuration.timeUnit=hour).price
          let hours: Integer= self.reservation.AssignedCar.
            oclAsType(Prepared).actualTime -
          self.reservation.beginning.Time()).floor()
          in
          self.oclAsType(ApologiseForLatePreparation).^apply()
          and reimburse(self.reservation.renter,
self.hours*self.hourlyPaid)
```

ApologisePlusCancelation

```
context   ApologisePlusCancelation:: apply() :  Boolean
post:
          self.oclAsType(ApologiseForLatePreparation).^apply()
          and  self.reservation.oclIsKindOf(CanceledReservation)
```

RentalDetails

```
context   RentalDetails:: apply() :  carGroup
post:
          answerDetails=Tuple(beginning=self.rental.beginning,
          agreedEnding= self.rental.agreedEnding, pickUpBranch=
          self.rental.pickUpBranch.name,
rentersID=self.rental.renter.id,
            otherBadExp=self.rental.renter.faults->size()>1)
```

FreeCarsInNotPickedUpR

```
context   FreeCarsInNotPickedUpR:: apply() :  Set(Reservation)
post:
          RentalAgreement.allInstances()->select(r| r.pickUpBranch=
            self.branch and r.beginning=today())->reject(r|
            r@pre.oclIsKindOf(OpenedRental) or
            r@pre.oclIsKindOf(CanceledReservation)) ->forAll(r|
            r.oclIsKindOf(CanceledCompany) and
            r.oclAsType(CanceledCompany).motivation=
            CancellingMotivation::no-show and
            r.oclIsKindOf(GuaranteedReservation) implies
          charge(r.oclAsType(GuaranteedReservation).creditCardNumber,
            r.oclAsType(GuaranteedCanceled).fine)
```

NonReturnedRentals

```
context   NonReturnedRentals:: apply() :  carGroup
post:
          self.answerRentals=RentalAgreement.all
          Instances->select(r|r.returnBranch=self.branch and
```

```
          r.agreedEnding=today() and
not(r.oclIsKindOf(CanceledReservation))
             and not(r.oclIsKindOf(ClosedRental))) and
          self.answerRentals->forAll(r|rd.oclIsNew() and
            rd.oclIsKindOf(RentalDetails) and rd.rental=r)
```

NonReturned3DayRentals

```
context   NonReturned3DayRentals:: apply() :  Set(Reservation)
post:
          let rentals: Set(OpenedRental)= OpenedRental.all
          Instances()->select(r|r.returnBranch=self.branch and
          not(self.oclIsKindOf(ClosedRental)) and not
            self.oclIsTypeOf(CanceledReservation) and today()-
            self.agreedEnding.date() =3)
          in
          Sequence{1..rentals->size()}->forall(i|answerDetails->at(i)=
          Tuple(beginning=rentals->at(i).beginning, agreedEnding=
            rentals->at(i).agreedEnding, pickUpBranch=
            rentals->at(i).pickUpBranch.name, rentersID=
            rentals->at(i).renter.id, rentersName
            =rentals->at(i).renter.name,rentersTelephone=
            rentals->at(i).renter.telephone)
```

CarReturn

```
context   CarReturn:: apply() :  Set(Reservation)
post:
          let closedR:closedRental=
self.rental.oclAsType(ClosedRental)
          in
          let lateRCost:Money= self.closedR.rentalPriceWithTax+
          self.closedR.oclAsType(LateReturn).extraCostWithTax
          let dropPenalty: Boolean= self.rental.returnBranch<>
          self.rental.actualReturnBranch
          in
          self.rental.oclIsKindOf(ClosedRental) and
            self.closedR.actualReturn= now() and
            self.closedR.actualReturnBranch=self.branch and
            self.closedR.actualReturnBranch<> self.closedR.pickUpBranch
          implies self.branch.car->includes(self.closedR.car) and
          if (self.closedR.oclIsKindOf(LateReturn)) then
            FaultSeriousness.allInstances-
>exists(fs.badExperience.type=
            lateReturn and fs.closedRental=self.closedR and fs.degree=
            degree(self.rental.oclAsType(LateReturn).extraInterval))
and
            not self.closedR.driver.oclIsTypeOf(LoyaltyMember) and
             if dropPenalty then
               self.answerCost=self.lateRCost+dropOffPenalty()
            else
               self.answerCost=self.lateRCost
            end if
          else
             if dropPenalty then
               self.answerCost=self.closedR.rentalPriceWithTax+
          dropOffPenalty()
            else
               self.answerCost=self.closedR.rentalPriceWithTax
```

```
            end if
          end if
```

PaymentData

```
context   PaymentData:: apply() :  Money
post:
          self.rental.paymentType=self.payType and
          self.rental.creditCarNumberDamages=self.creditCardDamages
```

CAR MANAGEMENT EVENTS

MoveCars

```
context   MoveCars:: movedCars() :  Boolean
post:
          if self.neededCars>self.carsOtherBranchCanMove then
             result=self.carsOtherBranchCanMove
          else
             result=neededCars


context   MoveCars:: apply() :  Boolean
post:
          self.answerNeededCars=self.neededCars-self.movedCars
```

ReceiveCar

```
context   ReceiveCar:: apply() :  CarModel
post:
          c.oclIsNew() and c.oclIsKindOf(OwnCar) and
          c.registrationNumber=self.regNumber and
self.currentMileage=0 and
            self.mileageFromLastService=0 and
            self.lastMaintenanceDate=today() and
            self.acquisitionDate=today() and self.branch.car-
      >includes(c)
```

RequestTransfer

```
context   RequestTransfer:: apply() :  Boolean
post:
          self.oclAsType(MoveCars).^apply() and self.otherBranch.
            carsAvailable@pre->intersection(self.otherBranch.car->
            select(c|c.oclIsKindOf(BeingTransferredCar) and
            c.oclAsType(BeingTransferredCar).destination=
          self.askingBranch))->size()=movedCars
```

DoTransfer

```
context   DoTransfer:: apply() :  Boolean
post:
          self.oclAsType(MoveCars).^apply() and self.askingBranch.
            carsAvailable@pre->intersection(self.askingBranch.car->
```

```
            select(c|c.oclIsKindOf(BeingTransferredCar) and
            c.oclAsType(BeingTransferredCar).destination=
          self.otherBranch))->size()=movedCars
```

CalculateOwnCars

```
context   CalculateOwnCars:: apply() :  Boolean
post:
          answerLack=self.ownCars< 1.1*self.quota and answerSuperplus=
             self.ownCars >1.1*self.quota and carsAvailable>0


context   CalculateOwnCars:: ownCars() :  Boolean
post:
          result=self.branch.car->select(c|c.oclIskindOf(OwnCar)
          and c.carGroup=self.carGroup)->size()


context   CalculateOwnCars:: quota() :  Boolean

post:
          result=self.branch.carGroupQuota->select(cGQ|
            cgQ.carGroup= self.carGroup).quota


context   CalculateOwnCars:: carsAvailable() : Natural
post:
          self.branch.groupAvailability->select(ga|
            ga.carGroup=self.carGroup).quantity
```

WithSurplus

```
context   WithSurplus:: allInstances() :  Boolean
post:
          CalculateOwnCars.allInstances()->select(c|c.answerSurplus)


context   WithSurplus:: apply() :  Boolean
post:
          self.oclAsType(CalculateOwnCars).^apply() and
          self.answerSurplusQ= self.carsAvailable.min(self.ownCars
            -self.quota*1.1)
```

WithLack

```
context   WithLack:: allInstances() :  Boolean
post:
          CalculateOwnCars.allInstances()->select(c|c.answerLack)


context   WithLack:: apply() :  Boolean
post:
          self.oclAsType(CalculateOwnCars).^apply() and
          self.answerSurplusQ=self.quota*1.1- self.ownCars
```

BranchesWithSurplusOfCarGroup

```
context   BranchesWithSurplusOfCarGroup:: apply() :  Natural
post:
          let own(b:branch)=b.car->select(c|c.oclIskindOf(OwnCar) and
            c.carGroup=self.carGroup)->size()
          let quant(b:Branch,cg:carGroup):Natural= GroupAvailability.
            allInstances()->select(ga|ga.branch=b and
            ga.carGroup=cg).quantity
          let CGquota(b:Branch,cg:carGroup):Natural= CarGroupQuota.
            allInstances->select(ga|ga.branch=b and
      ga.carGroup=cg).quota
          in
          let branches: Set(Branch)=self.branch.receiver->
          select(quant(b,self.carGroup)>0 and
own(b)<CGquota(b,self.car))
          in
          Sequence{1..branches->size()}->forall(i|answersTransfers->
            at(i)=Tuple(branch=branches->at(i), numCars=
            (CGquota(branches->at(i),self.car)-
            quant(branches->at(i).self.car)))
```

BranchesWithLackOfCarGroup

```
context   BranchesWithLackOfCarGroup:: apply() :  Natural
post:
          let own(b:branch)=b.car->select(c|c.oclIskindOf(OwnCar) and
            c.carGroup=self.carGroup)->size()
          let quant(b:Branch,cg:carGroup):Natural= GroupAvailability.
            allInstances()->select(ga|ga.branch=b and
            ga.carGroup=cg).quantity
          let CGquota(b:Branch,cg:carGroup):Natural= CarGroupQuota.
            allInstances()->select(ga|ga.branch=b and ga.carGroup=cg)
            .quota
          let branches: Set(Branch)=self.branch.receiver->
          select(own(b)>CGquota(b,self.car))
          in
          Sequence{1..branches->size()}->forall(i| answersTransfers->
            at(i)=Tuple(branch=branches->at(i), numCars=
            (own(branches->at(i).self.car) -
          CGquota(branches->at(i).self.car)))
```

OrderCar

```
context   OrderCar:: apply() :  PendantCarOrder
post:
          pco.oclIsNew() and pco.oclIsKindOf(PendantCarOrder)
          and self.branch->includes(pco) and pco.id=
            PendantCarOrder.getNewId()
```

SellCar

```
context   SellCar:: notAssigned() :      Boolean
post:
          result=self.Car.RentalAgreement->select(ra|
            ra.oclIsTypeOf(OpenedRental)->isEmpty()
```

```
context   SellCar:: apply() :  Boolean
post:
          self.car.oclIsTypeOf(ToBeSoldCar)
```

ConfirmCarSale

```
context   ConfirmCarSale:: carToBeSold() :     Boolean
post:
          result = self.Car.oclIsTypeOf(ToBeSoldCar)


context   ConfirmCarSale:: apply() :  Boolean
post:
          self.Car.oclIsTypeOf(SoldCar) and self.Car.oclAs
          Type(SoldCar).disposalDate=today()
```

BRANCH, CAR GROUP AND MODELS MANAGEMENT EVENTS

ExistingCountryCheck

```
context   ExistingCountryCheck:: apply() :  Set(BranchType)
post:
          let count: Set(Country)=Country.allInstances()->
          select(c|c.name=self.countryName)
          in
          self.answerCountryExists=count->notEmpty() and
          count->notEmpty implies self.answerCountry=count->any()
```

RecordCountryInformation

```
context   RecordCountryInformation:: apply() :  Set(BranchType)
post:
          c.oclIsNew() and c.oclIsTypeOf(Country) and
          c.name=self.name and c.mechanicalConditionsRequirements=
            self.mechanicalCondReq and c.emissionsReq=self.emissionsReq
      and
            c.carTax=self.carTax and answerCountry=c
```

CreateBranch

```
context   CreateBranch:: branchesExists() :    Boolean
post:
          self.quotas->forAll(q|CarGroup->allInstances()->
            exists(cg|cg.name=q.carGroupName))


context   CreateBranch:: apply() :  Boolean
post:
          br.oclIsNew() and br.oclIsTypeOf(Branch) and
          br.name=self.branchName and br.branchType=BranchType.
          allInstances()->select(bt|bt.name=self.branchType)
          and self.quotas->forAll(q|br.carGroupQuota->
          includes(cgq|cgq.oclIsNew() and
cgq.oclIsTypeOf(CarGroupQuota)
          and cgq.carGroup=CarGroup.allInstances()->
          select(cg| cg.name=q.carGroupName) and cgp.branch=br)
```

CreateCarGroup

```
context   CreateCarGroup:: apply() :  Boolean
post:
          let previousCarGroup:CarGroup=CarGroup.allInstances()->
            select(cg|cg.name=self.previousGroupName)
          in
          cg.oclIsNew() and cg.oclIsTypeOf(CarGroup) and
          cg.name=self.carGroupName and
          if (self.previousCarGroup->notEmpty() ) then
            self.previousCarGroup.better=cg and cg.better=
            self.previousCarGroup.better@pre
          else -- is the worst
           cg.better=CarGroup.allInstances()->
            select(cg|cg.worse@pre->isEmpty())
          end if
          --assign default quota of the new car group to all branches
          and Branch->allInstances()->forall(b|cgq.oclIsNew()
          and cgq.oclIsTypeOf(CarGroupQuota) and
          cgq.quota=self.defaultQuota and cgq.carGroup=cg
          and answerCarGroup=cg
```

CreateCarModel

```
context   CreateCarModel:: apply() :  Boolean
post:
          self.answerModel.oclIsNew() and self.answerModel.name=
          self.name and self.answerModel.characteristics=techCars
```

ExistingCarModelCheck

```
context   ExistingCarModelCheck:: apply() :  Boolean
post:
          let carM:Set(CarModel)= CarModel.allInstances()->
            select(cM|cM.name=self.modelName)
          in
          self.answerCarModelExists= self.carM->notEmpty() and
          self.answerModel=self.carM->any()
```

AddModelToGroup

```
context   AddModelToGroup:: apply() :  Boolean
post:
          self.group.carModel->includes(self.model)
```

ChangeModelGroup

```
context   ChangeModelGroup:: apply() :  Boolean
post:
          self.carG.carModel->includes(self.carM)
```

ChangeCarGroupQuota

```
context   ChangeCarGroupQuota:: apply() :  Set(Reservation)
```

```
post:
          self.branch.carGroupQuota->any(cgq| cgq.carGroup=self.carG).
            quota=self.newQuota
```

PERFORMANCE INDICATORS EVENTS

ExistingCountryBranchType

```
context   ExistingCountryBranchType:: CountryBranchTypeExists() :
Boolean
post:
          result=CountryBranchType.allInstances()->exists(cbt|
          cbt.branchType.name=self.branchType and
          cbt.country.name=self.countryName)


context   ExistingCountryBranchType:: apply() :  Boolean
post:

self.answerCountryBranchType=CountryBranchType.allInstances()
            ->select(cbt|cbt.branchType.name=self.branchType and
            cbt.country.name=self.countryName)
```

PerfomanceIndicatorCheck

```
context   PerfomanceIndicatorCheck:: apply() :  Boolean
post:
          let perfInd:Set(PerformanceIndicator)=PerformanceIndicator.
            allInstances()->select(pi|pi.name=self.name)
          in
          self.answerPerformanceIndExist=perfInd->notEmpty() and
           perfInd->notEmpty() implies
          self.answerPerformanceIndicator= perfInd->any()
```

CreateIndicator

```
context   CreateIndicator:: apply() :  Boolean
post:
          pi.oclIsNew() and pi.oclIsTypeOf(PerformanceIndicator)
          and pi.name=self.name and pi.description=self.description
and
            self.answerPerformanceIndicator=pi
```

DefineTargetForConcreteIndicator

```
context   DefineTargetForConcreteIndicator:: notPreviousTarget() :
Boolean
post:
          result=self.perfIndicator.countryBranchType->
            excludes(self.countryBranchType)


context   DefineTargetForConcreteIndicator:: apply() :  Boolean
post:
          ci.oclIsNew() and ci.oclIsKindOf(ConcreteIndicator)
          and ci.countryBranchType=self.countryBranchType and
```

```
          ci.performanceIndicator=self.perfIndicator and
          ci.targetValue=self.target
```

ChangeTargetForConcreteIndicator

```
context   ChangeTargetForConcreteIndicator:: apply() :  Boolean
post:
          self.perfInd.targetValue=self.newTarget
```

8. SEQUENCE DIAGRAMS

Notation

To elaborate the sequence diagrams I have considered appropriate and interesting to adopt UML 2.0 style. The main contribution, respect UML 1.5, for this project needs is the definition of an easy-to-use and clear mechanism to represent *include* relationships from use cases (which as far as I am concerned there was not a standard notation for it before). Moreover, this mechanism is part of an homogeneous frame to represent loops, alternatives and exception among others, via the definition of a general frame of interaction (the combined fragment) and a descriptor of the specific type of interaction (the interaction operator).

However, UML 2.0 mechanisms have not been enough for the expressivity and clarity aimed for this project. As a consequence, a few non-standard extensions have been used trying to follow, as far as possible, UML 2.0 *style*. A note is attached the first time is used a non-standard structure.

The basic extension is an "interaction operator" *forEach* to allow defining a loop respect a set of elements and having in each iteration a current element to be used as an entry parameter in any of the events of the fragment.

Previous remarks

At the beginning of the events section, it was stated that events would be modelled as objects. This modelling decision has an effect also in the sequence diagrams appearance because instead of invoking so-called *system operations,* objects (the events) will be created. However, it should be noticed that the basic semantic or what both representations pretend is exactly the same.

Apart from this appearance change, the reader should be aware of the implicit parameter passing between events via the attributes of the object as explained in [EE]. Concretely, it is assumed that "entrance" attributes or restriction expressions can take its values from data entered by the actor, by an "*answer"* attribute of a previous event or by the variable bind to a forEach structure.

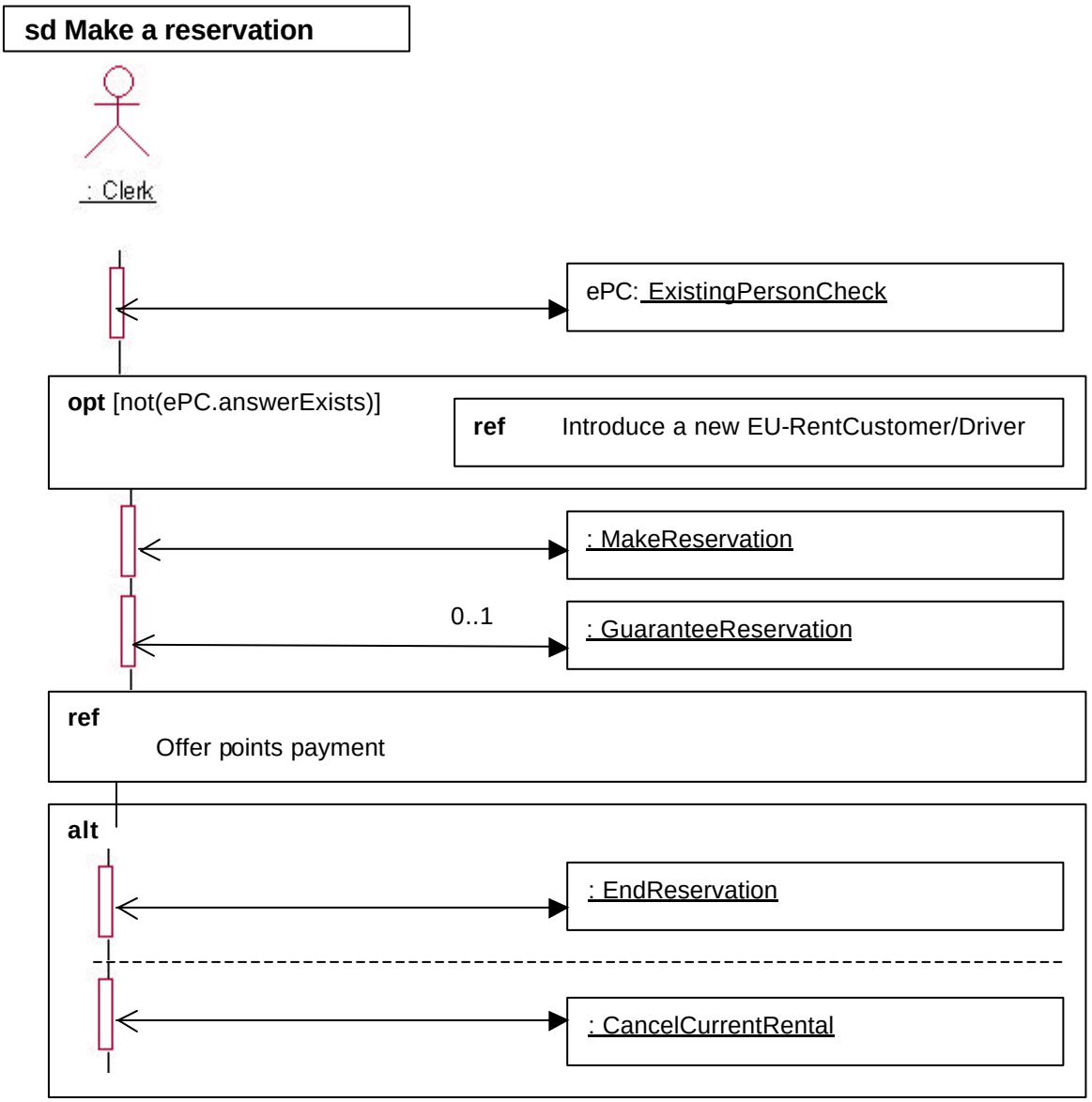
However, it is not clear how to concisely write an expression in a loop (or forEach) fragment where an attribute takes its value from a different instance in each iteration. In these cases, the instance to which the attribute belongs to has been omitted of the diagrams. It should be assumed that, this implicit instance corresponds to the event instance just created before the loop in the first iteration, and to the last event instance created in the loop for the rest of iterations (that is, what is logical).

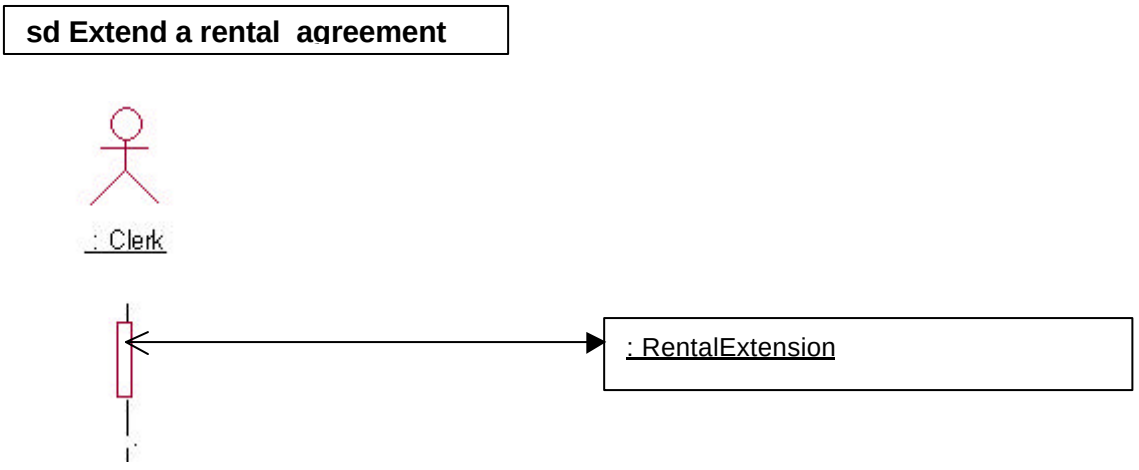
Additionally, in some cases difficulty has been found to establish *who* sparks off an event, belonging to a sequence of actions which are initially sparked off by a system user, but where user interaction is really scarcely needed (only if some non common conditions are achieved). Finally, the following convention has been decided: the first action will be shown to be sparked off by the user, will return the result to the system and the next events will be created by and returned to the system until user interaction is needed.

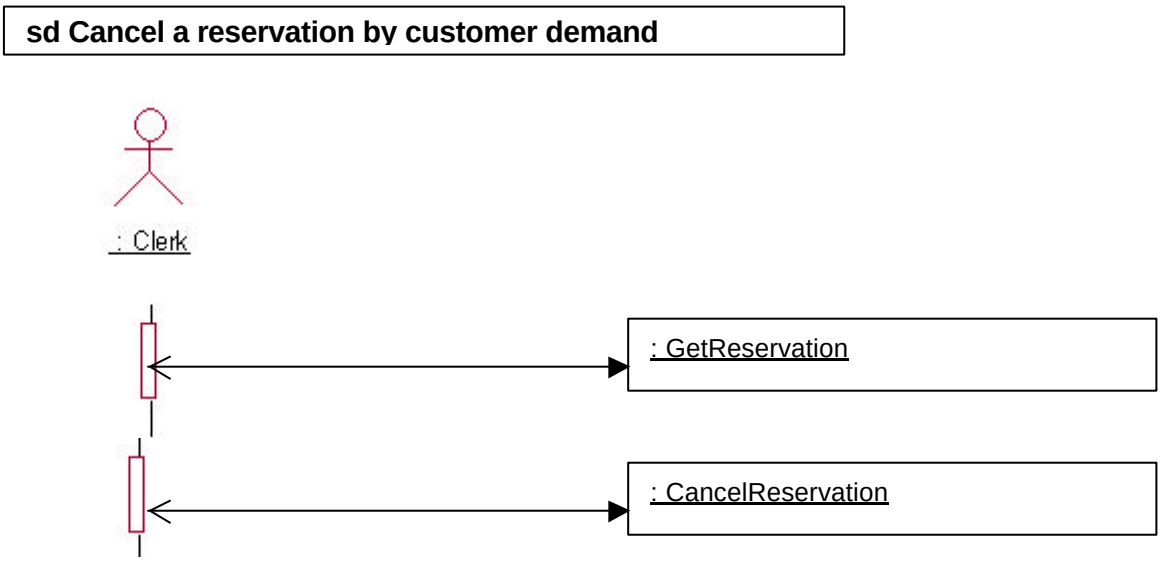
However, it is not clear how this logical convention is applied when the first action is inside a loop (or forEach structure). In this latter case, it has been decided that events

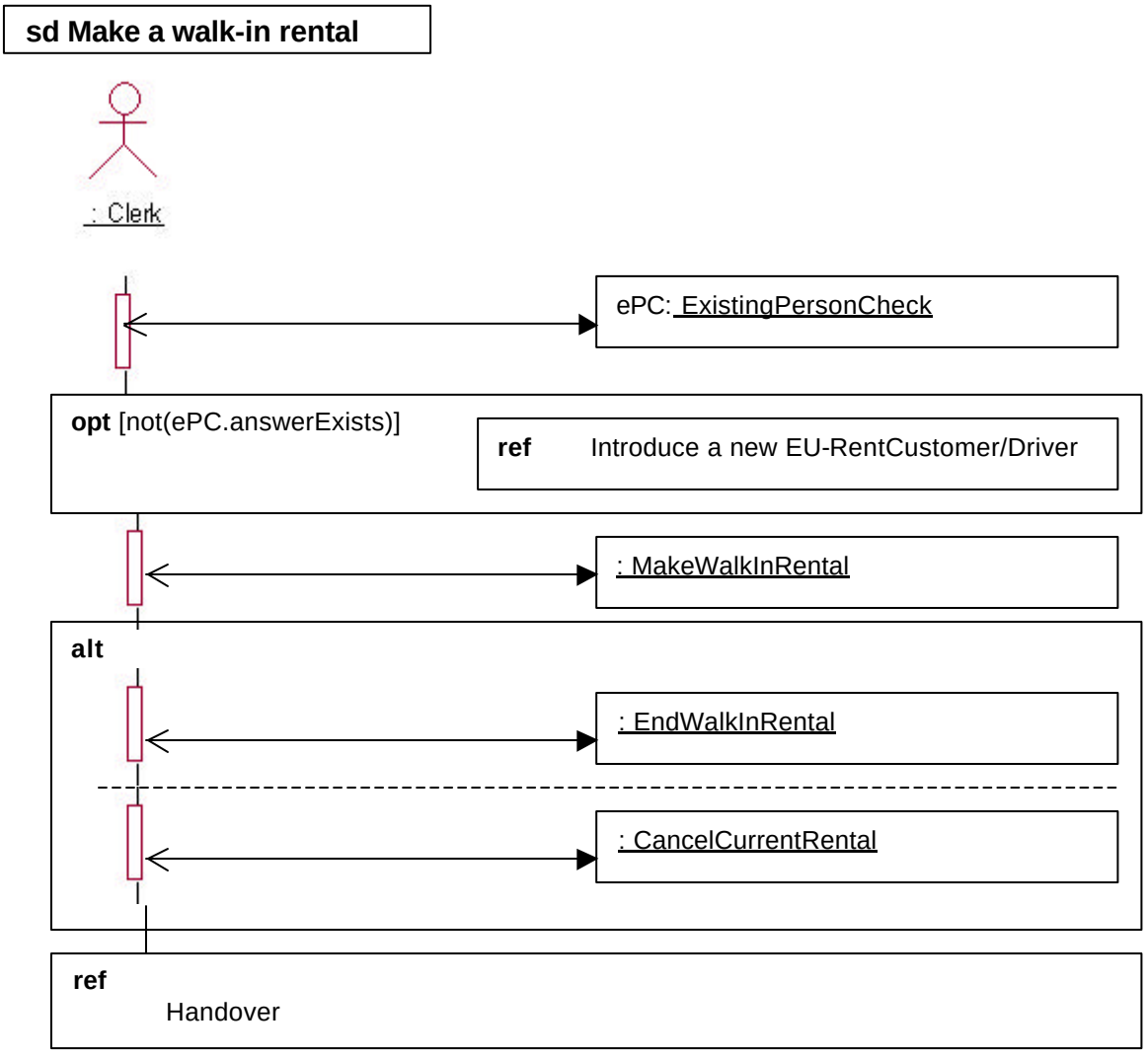
inside the loop are shown to be created by and returned to the system, unless user interaction needed. Moreover, to graphically show that the sequence of actions is sparked off by a system user, a non standard notation is used consisting in an arrow from the actor to the fragment.

Reservation management



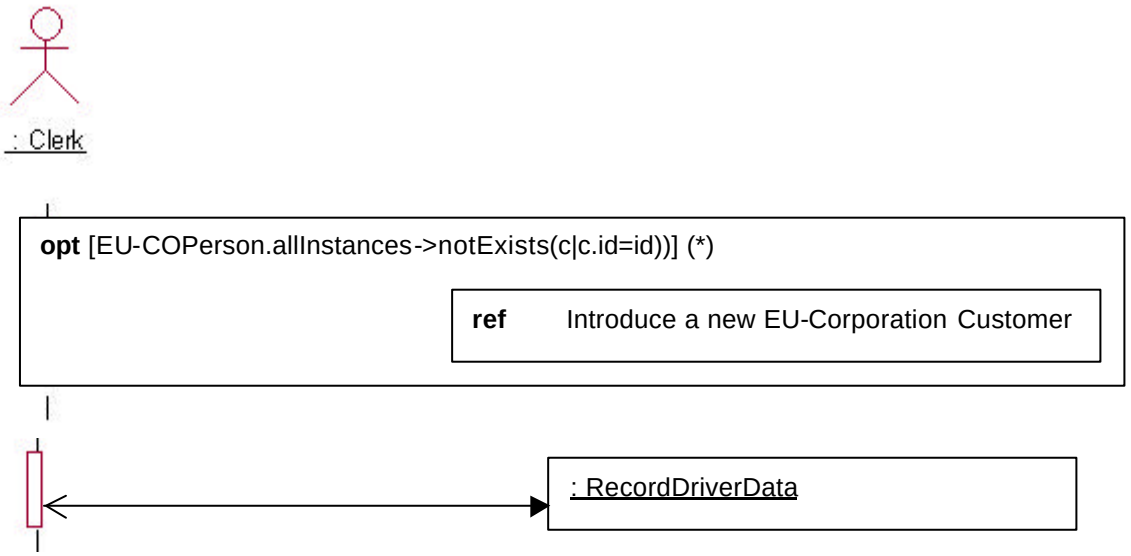



sd Cancel a reservation by customer demand
: Clerk
: GetReservation
: CancelReservation


sd Make a walk-in rental
: Clerk
ePC: ExistingPersonCheck
opt [not(ePC.answerExists)]
ref Introduce a new EU-RentCustomer/Driver
: MakeWalkInRental
alt
: EndWalkInRental
: CancelCurrentRental
ref
Handover

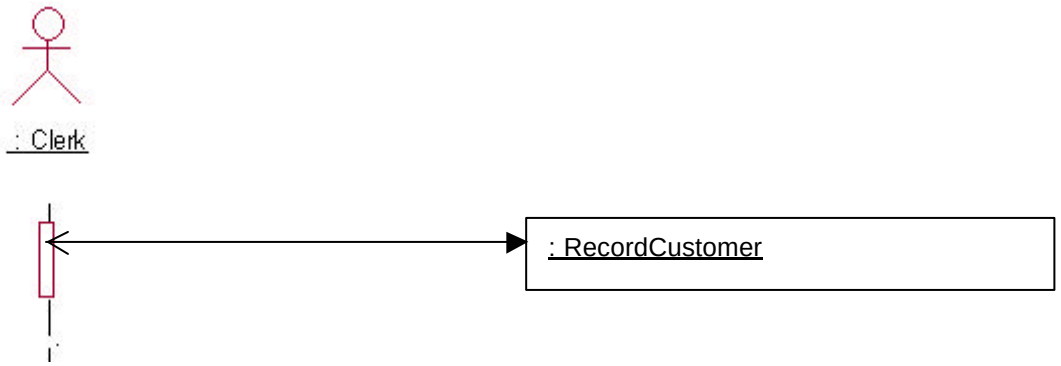
Customer management

sd Introduce a new EU-Rent customer/driver

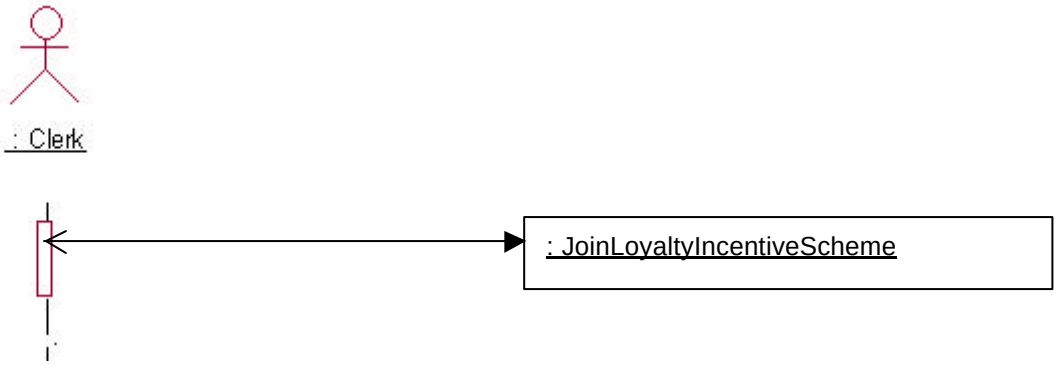


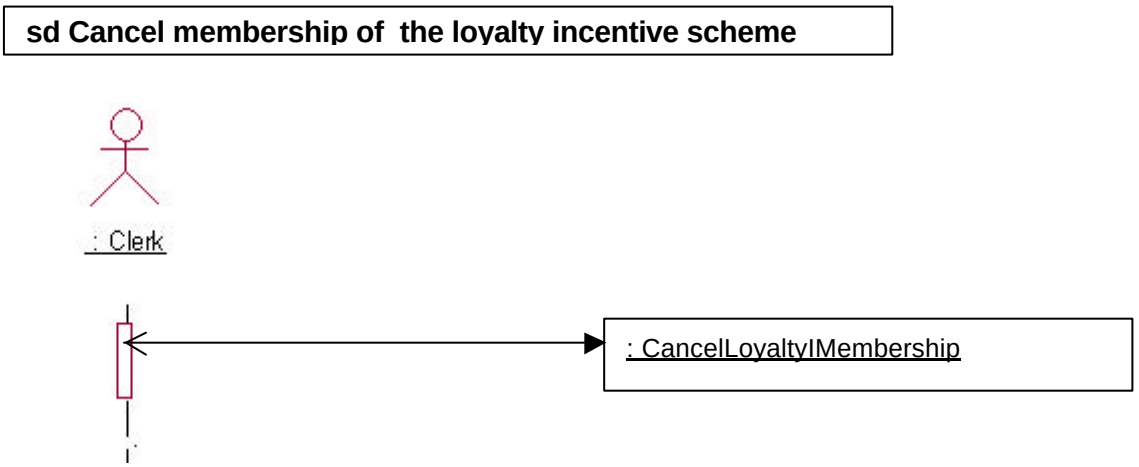
(*) We are assuming that there exists a class EU-CoPerson in the common information system of EU-Corporation, which encapsulates data of people who have had (or have) a contact with any of the companies of the corporation.

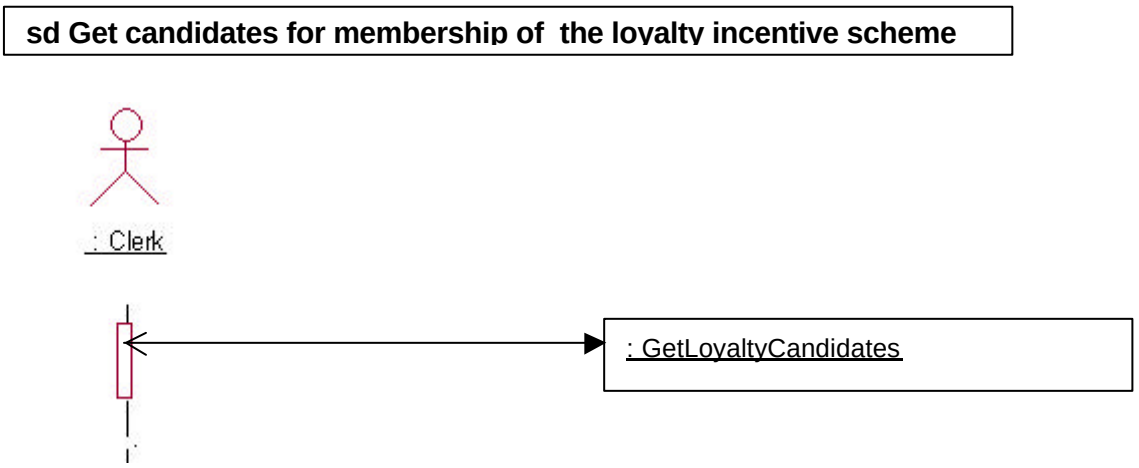
sd Introduce a new EU-Corporation Customer

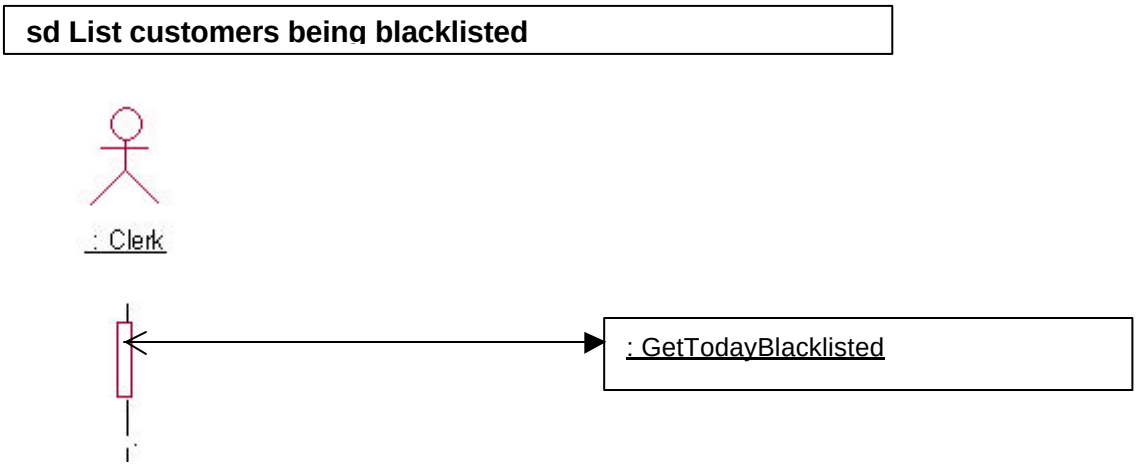


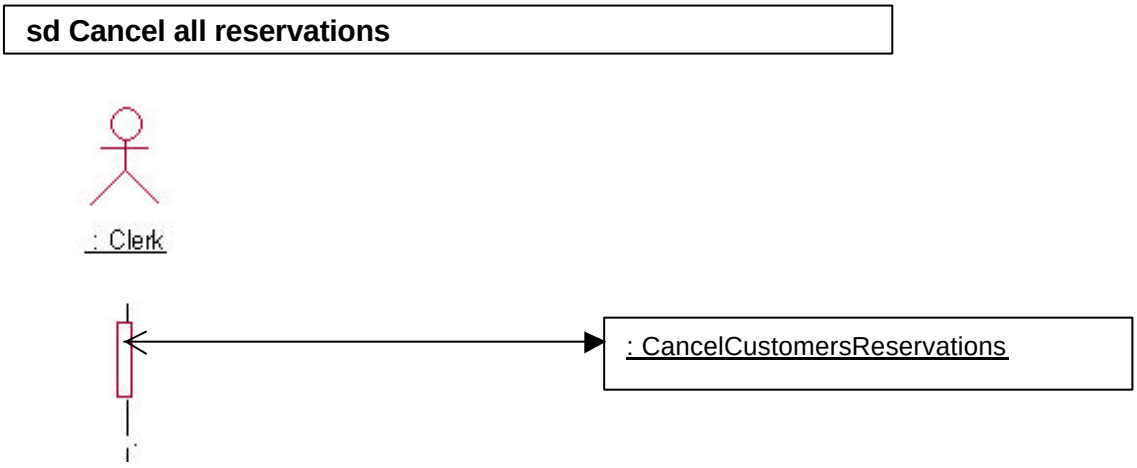
sd Join the loyalty incentive scheme

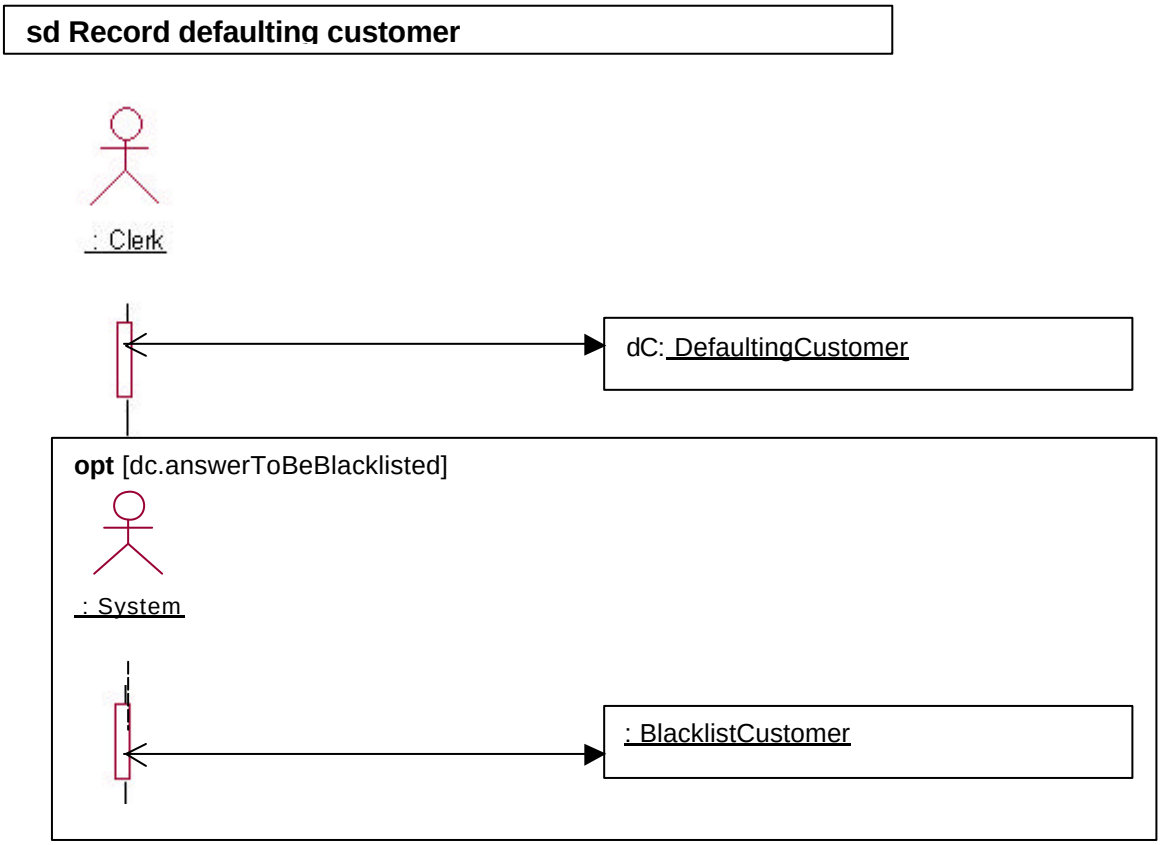



sd Cancel membership of the loyalty incentive scheme
: Clerk
: CancelLoyaltyIMembership

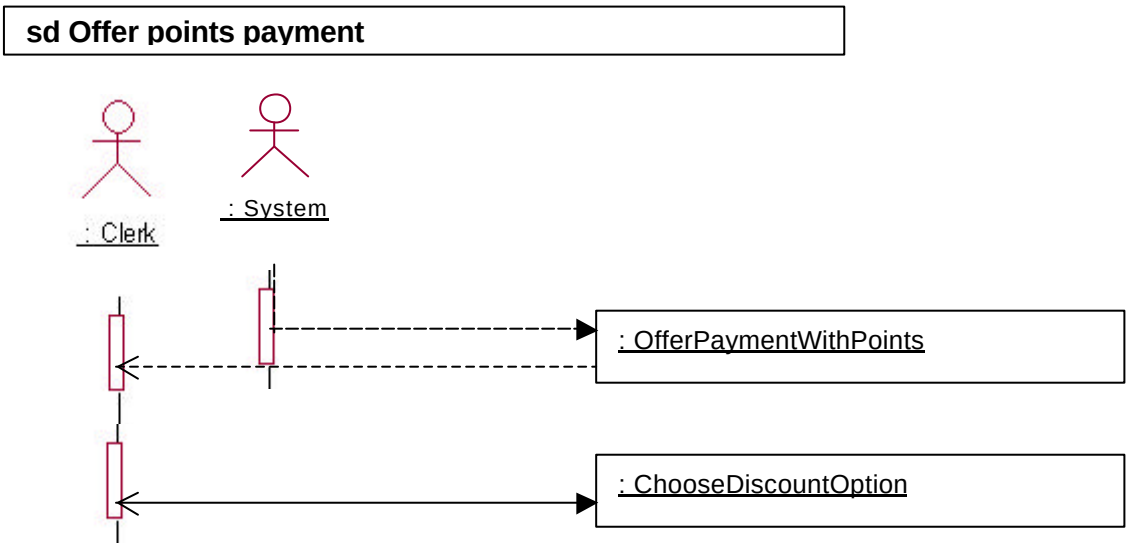

sd Get candidates for membership of the loyalty incentive scheme
: Clerk
: GetLoyaltyCandidates

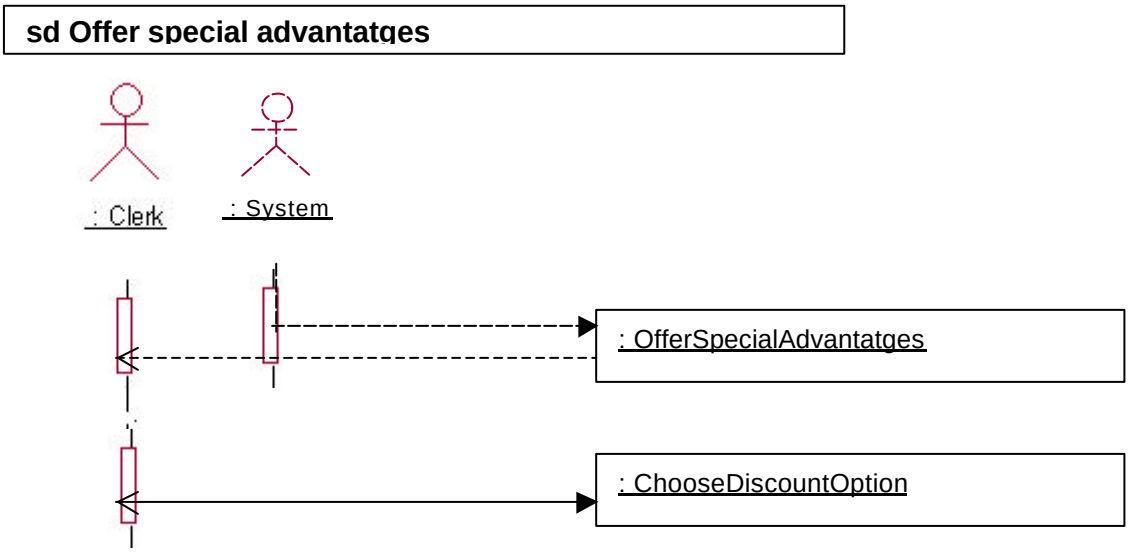

sd List customers being blacklisted
: Clerk
: GetTodayBlacklisted

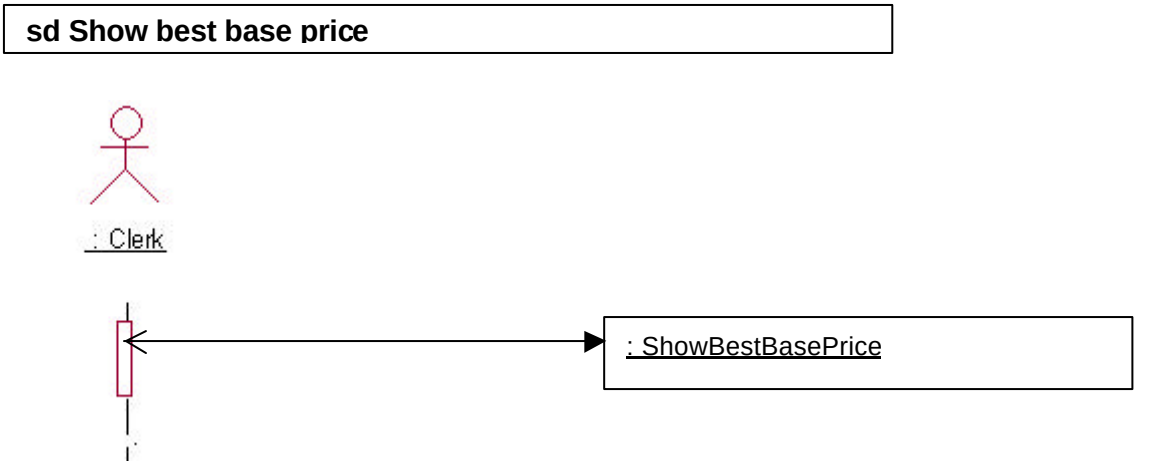

sd Cancel all reservations
: Clerk
: CancelCustomersReservations

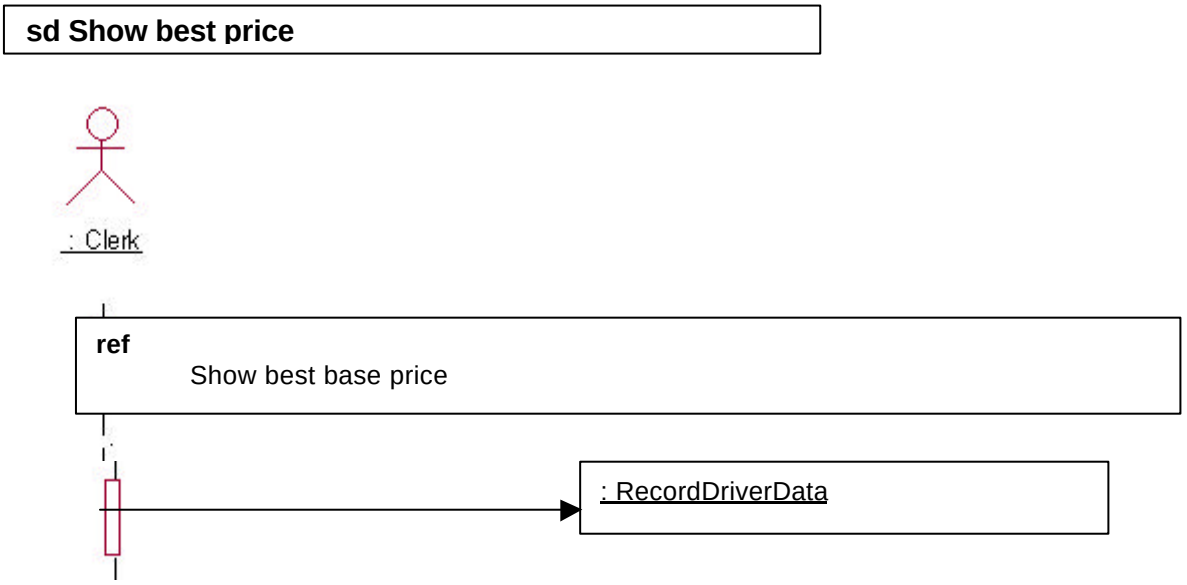

sd Record defaulting customer
: Clerk
dC: DefaultingCustomer
opt [dc.answerToBeBlacklisted]
: System
: BlacklistCustomer

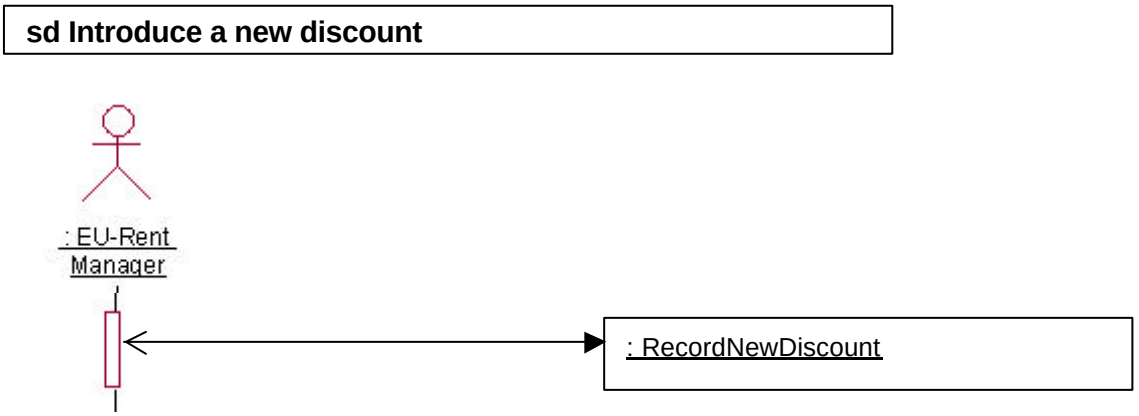
Pricing and discounting management

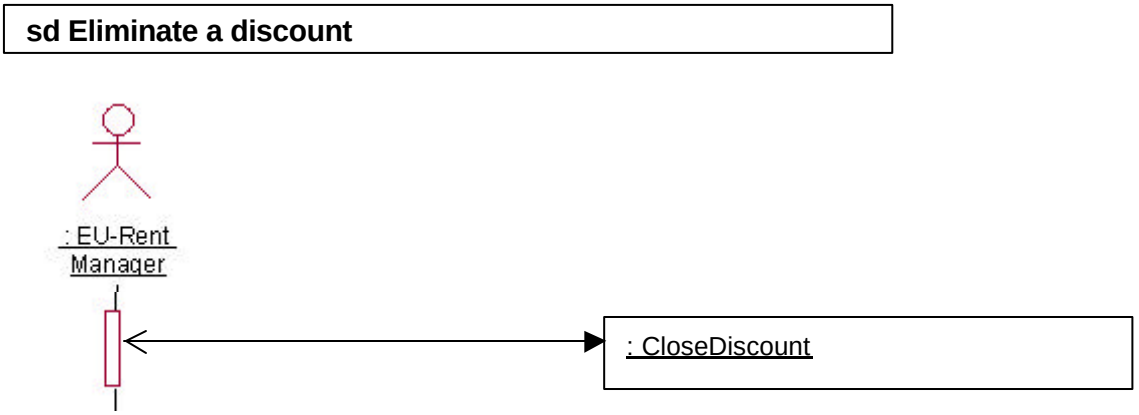




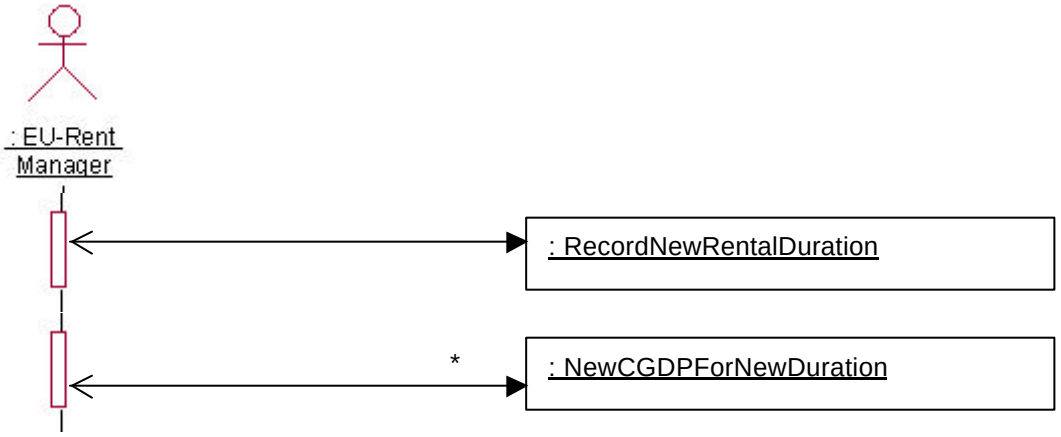



sd Show best price
: Clerk
ref
Show best base price
: RecordDriverData

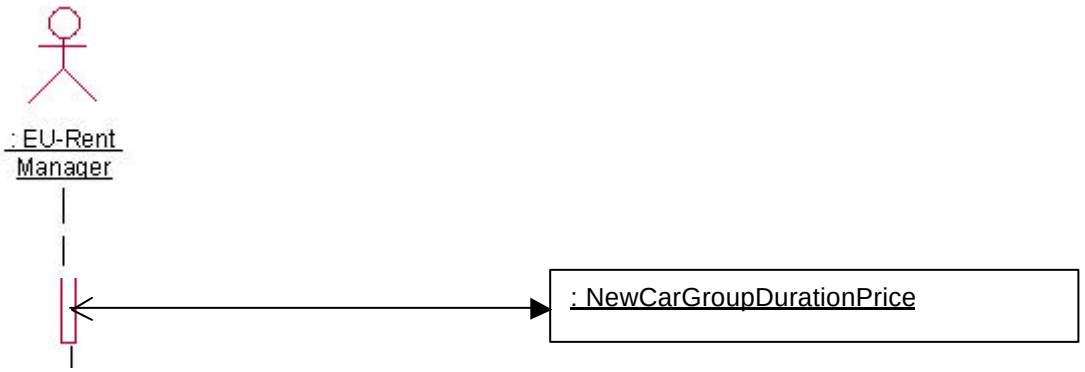

sd Introduce a new discount
: EU-Rent Manager
: RecordNewDiscount


sd Eliminate a discount
: EU-Rent Manager
: CloseDiscount

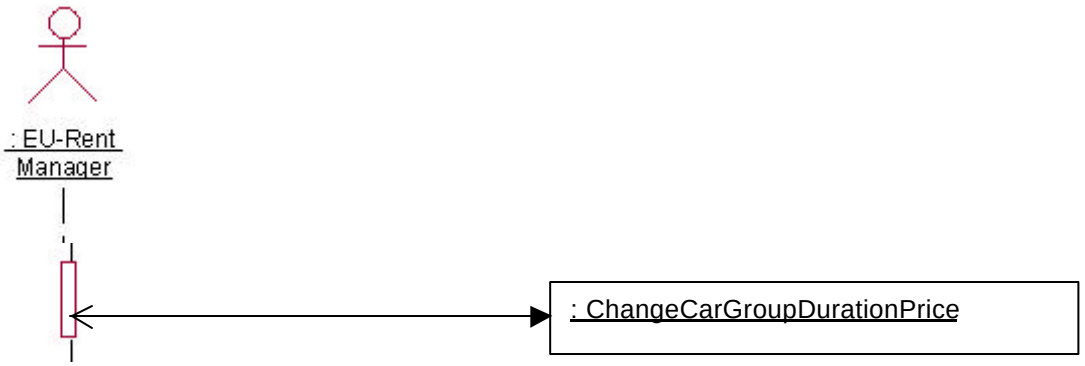
sd Create a new rental duration



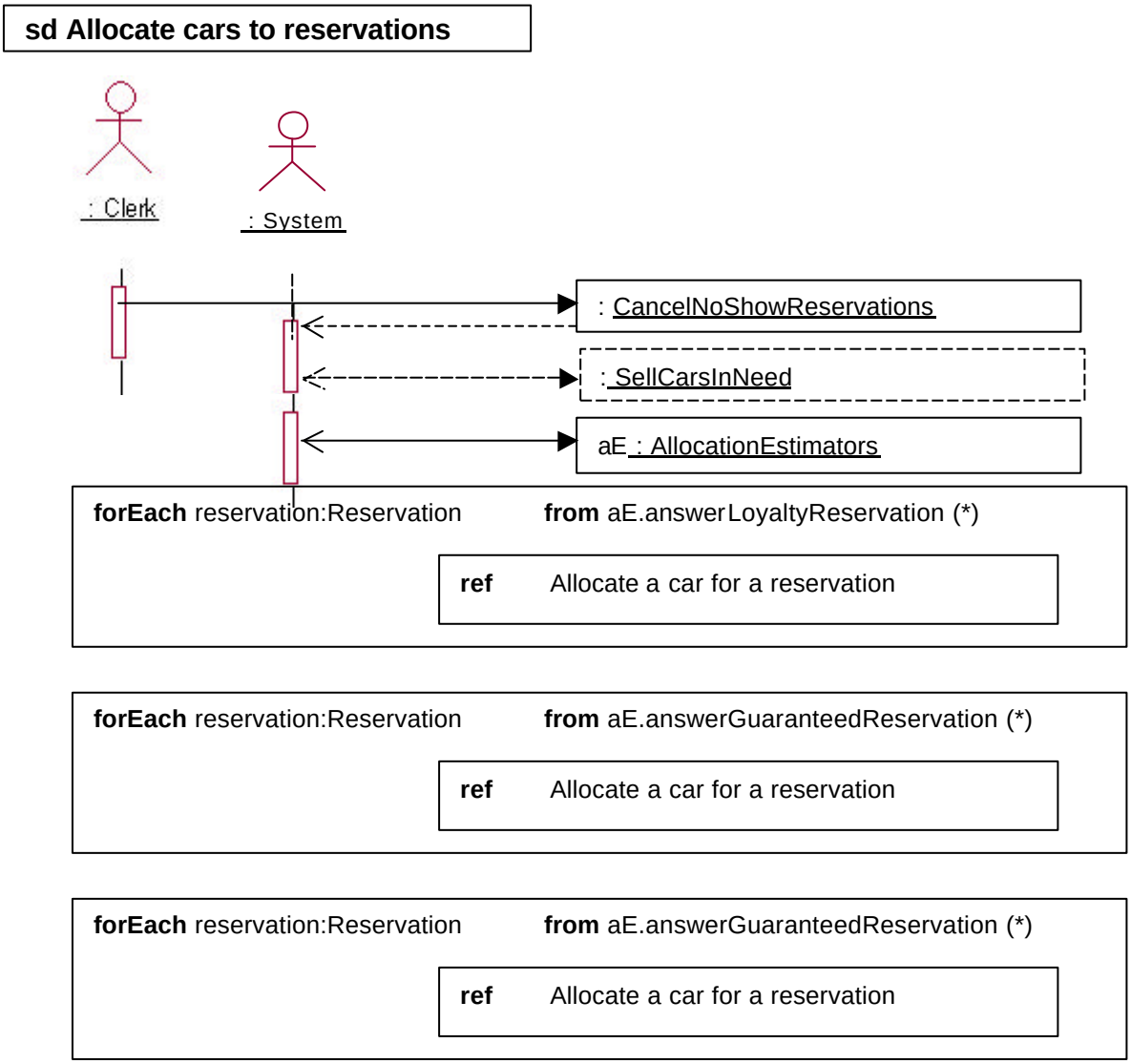
sd Create a car group duration price



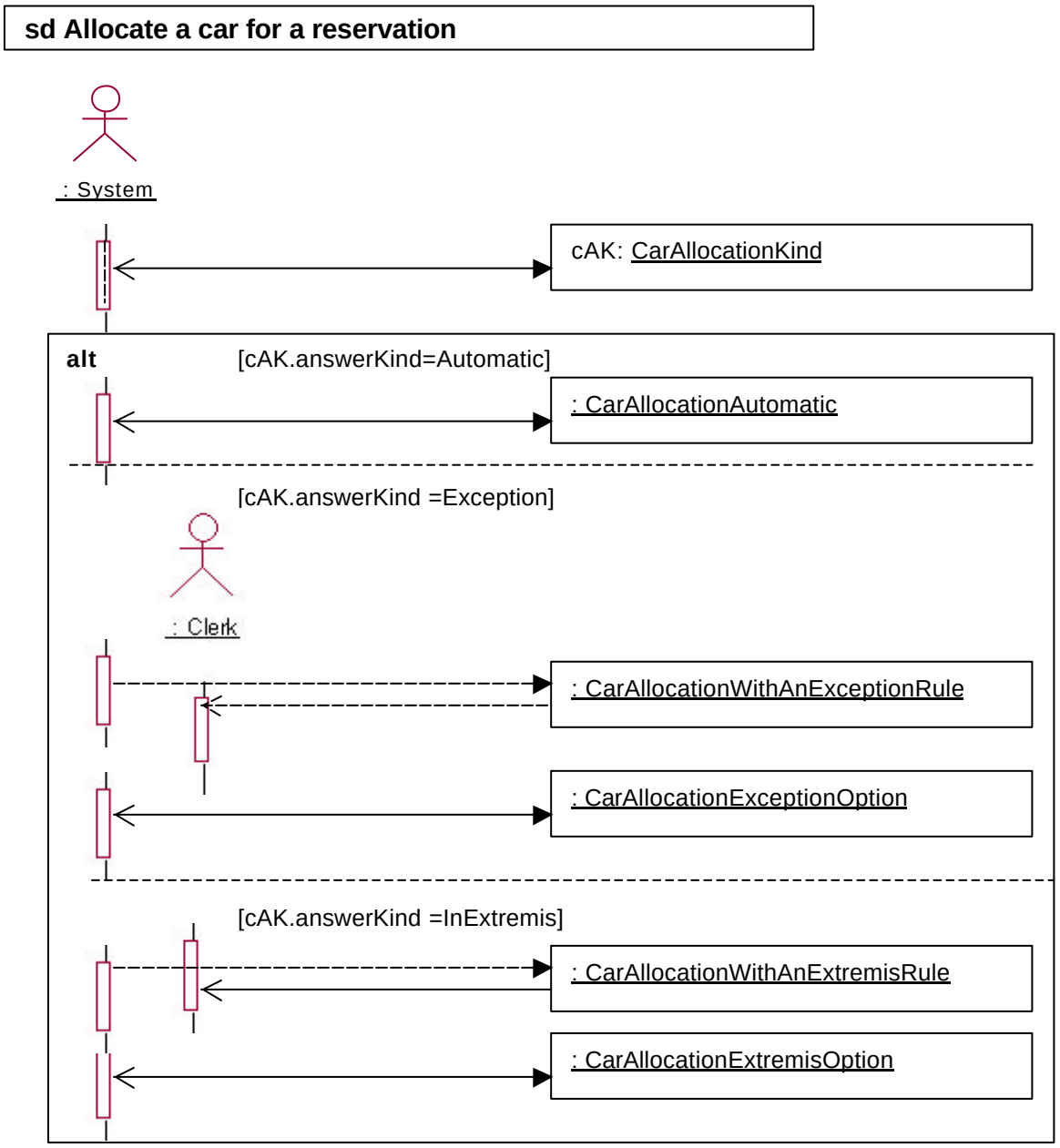
sd Change price for a car group duration price

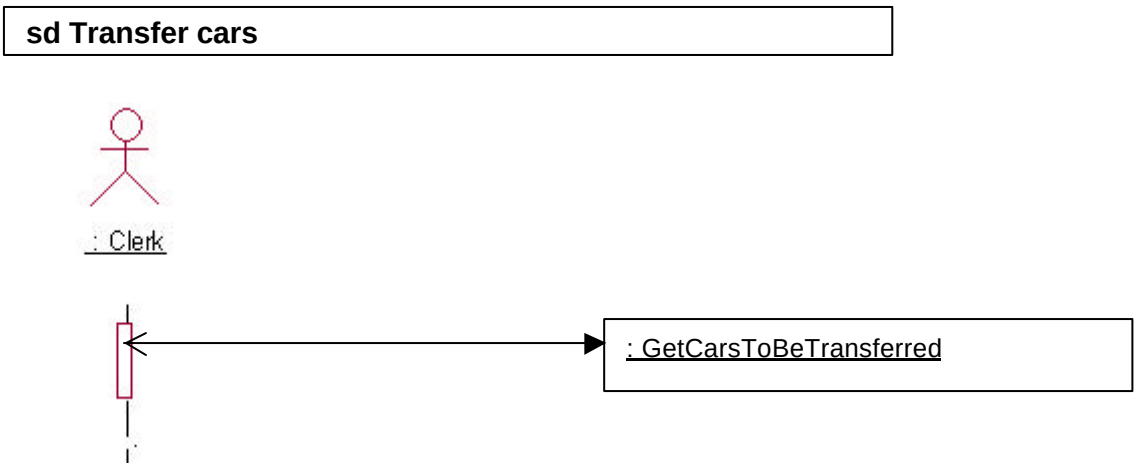


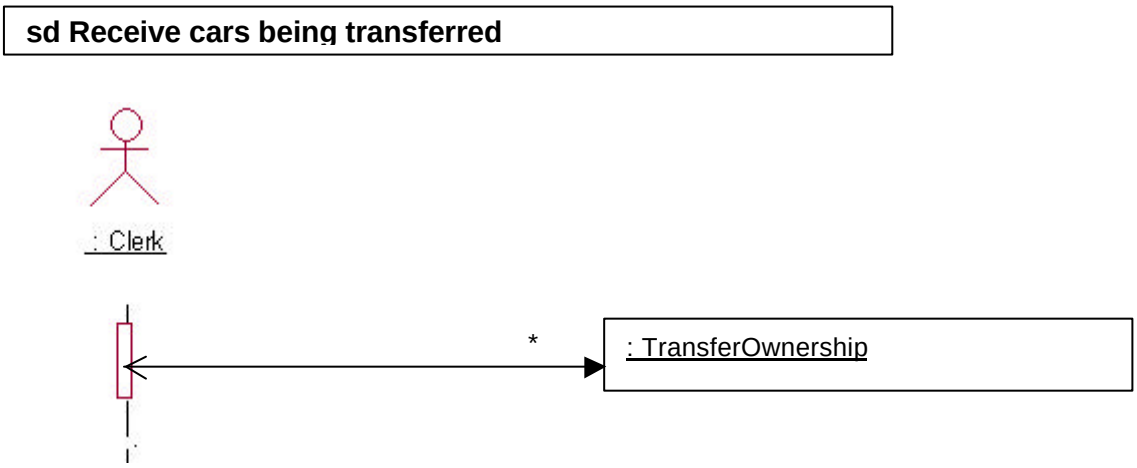
Car allocation

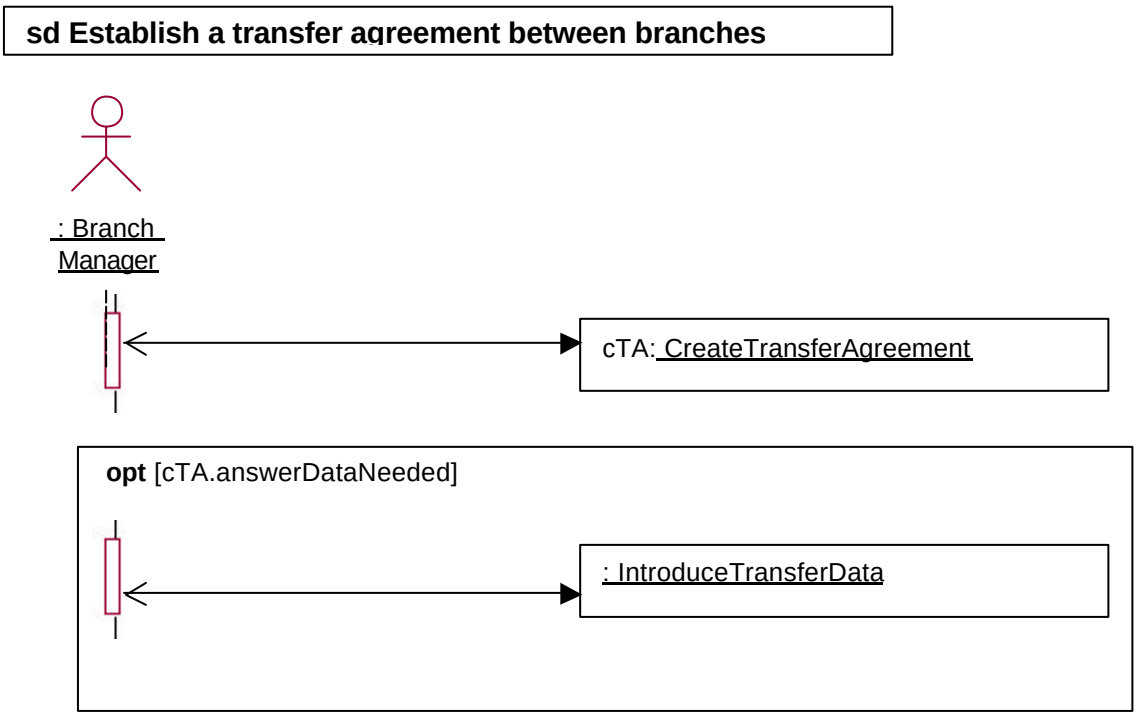


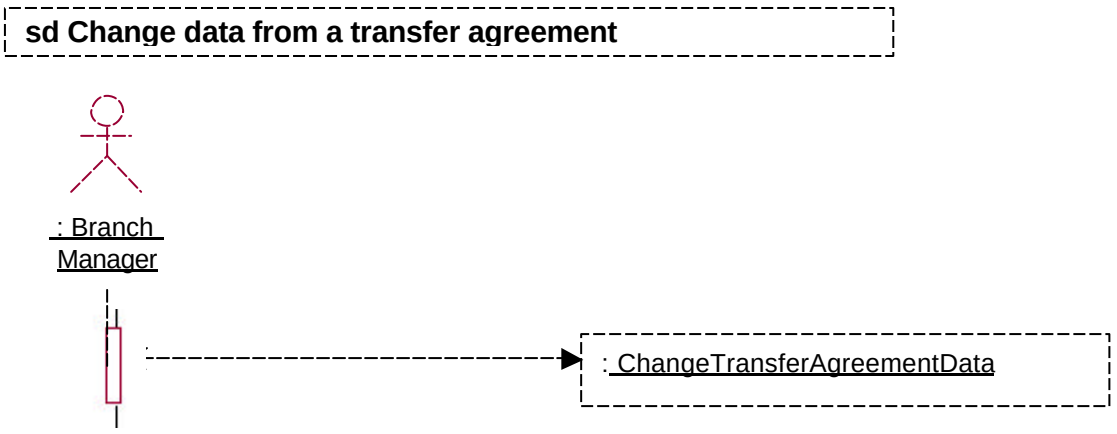
(*) Note the use of the non-standard UML 2.0 forEach as well as the use of the system actor.

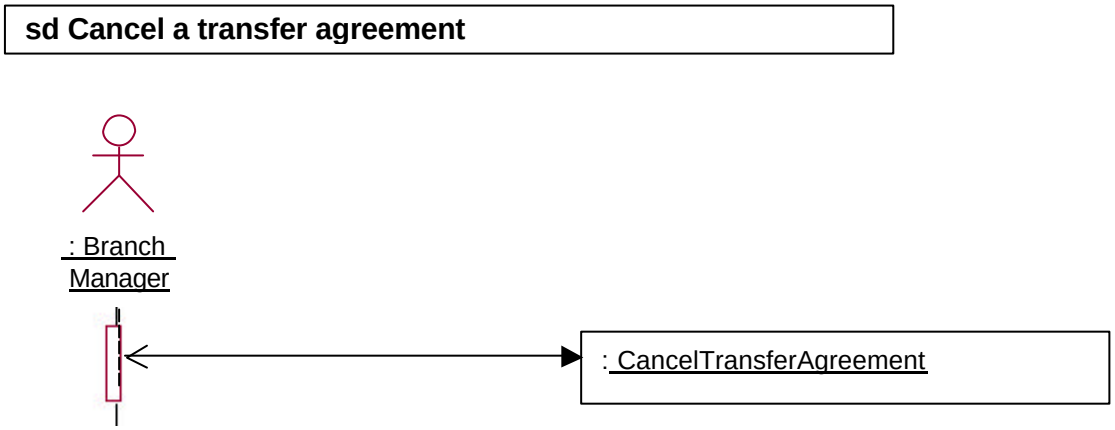

sd Allocate a car for a reservation
: System
cAK: CarAllocationKind
alt
[cAK.answerKind=Automatic]
: CarAllocationAutomatic
[cAK.answerKind =Exception]
: Clerk
: CarAllocationWithAnExceptionRule
: CarAllocationExceptionOption
[cAK.answerKind =InExtremis]
: CarAllocationWithAnExtremisRule
: CarAllocationExtremisOption


sd Transfer cars
: Clerk
: GetCarsToBeTransferred

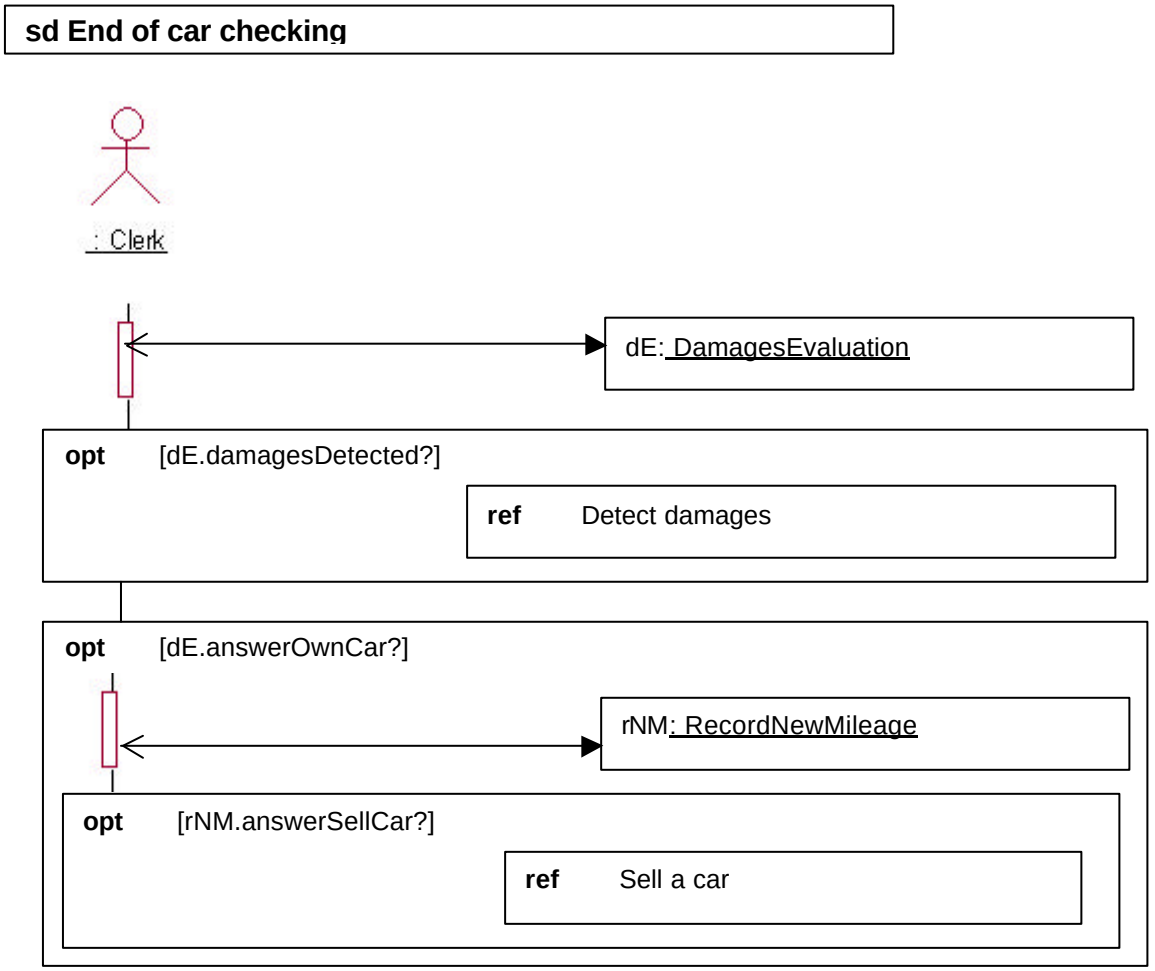

sd Receive cars being transferred
: Clerk
*
: TransferOwnership

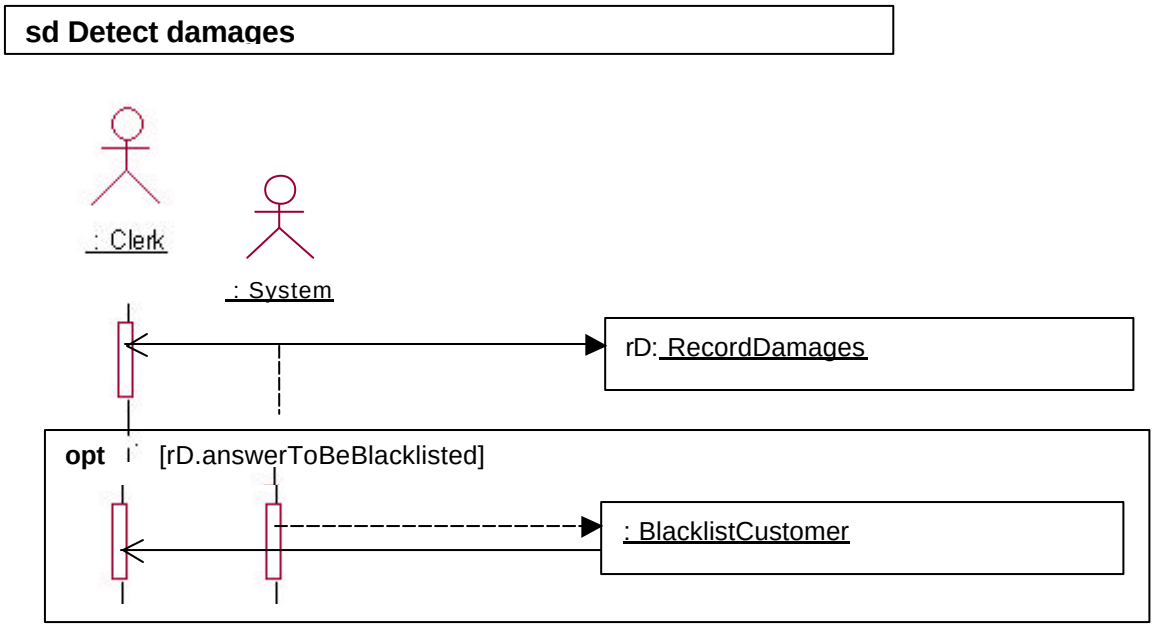

sd Establish a transfer agreement between branches
: Branch Manager
cTA: CreateTransferAgreement
opt [cTA.answerDataNeeded]
: IntroduceTransferData

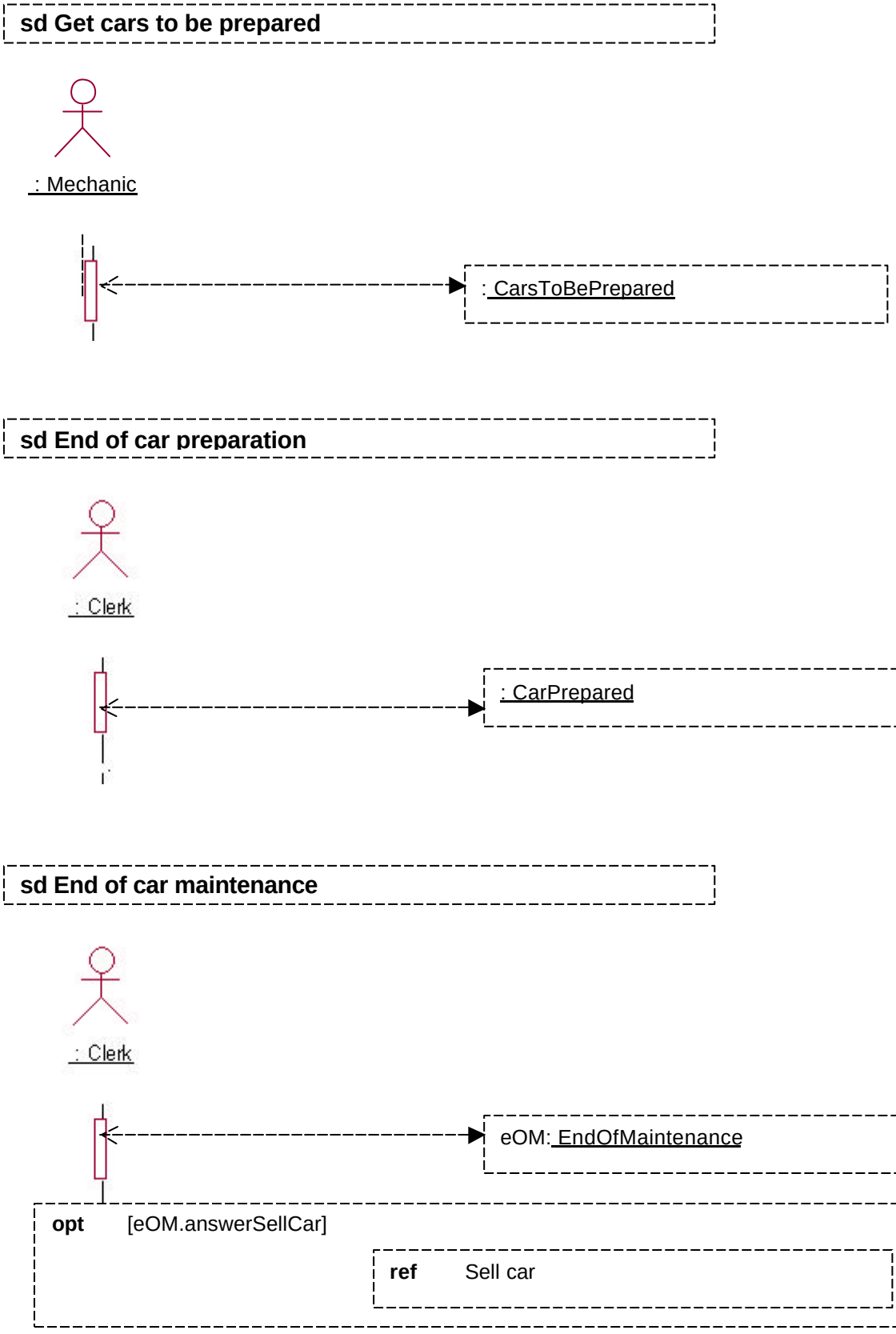

sd Change data from a transfer agreement
: Branch Manager
: ChangeTransferAgreementData

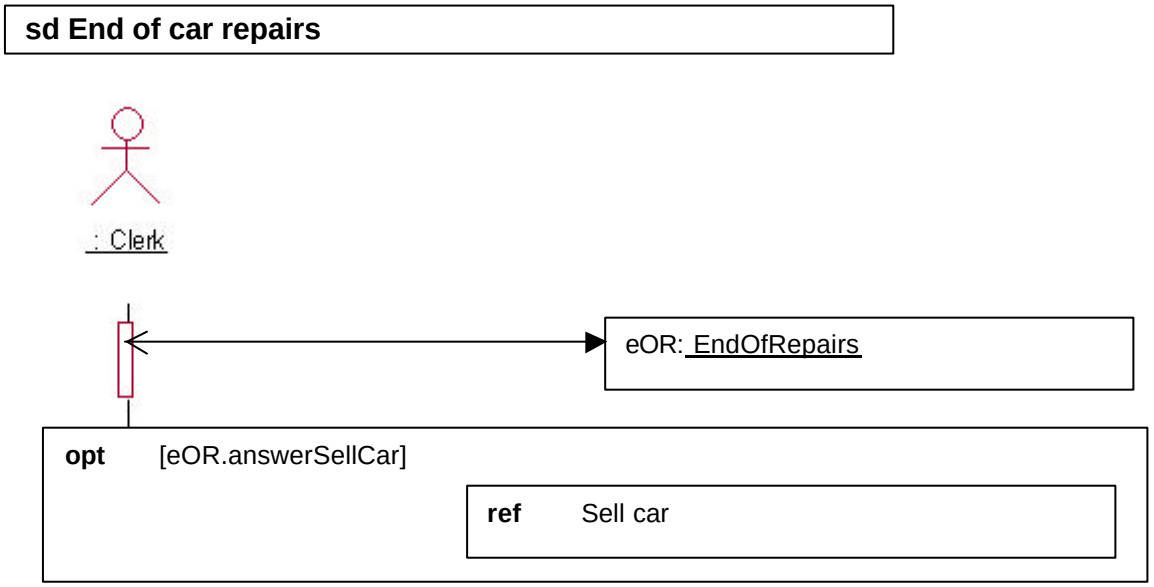

sd Cancel a transfer agreement
: Branch Manager
: CancelTransferAgreement

Car Preparation and maintenance management

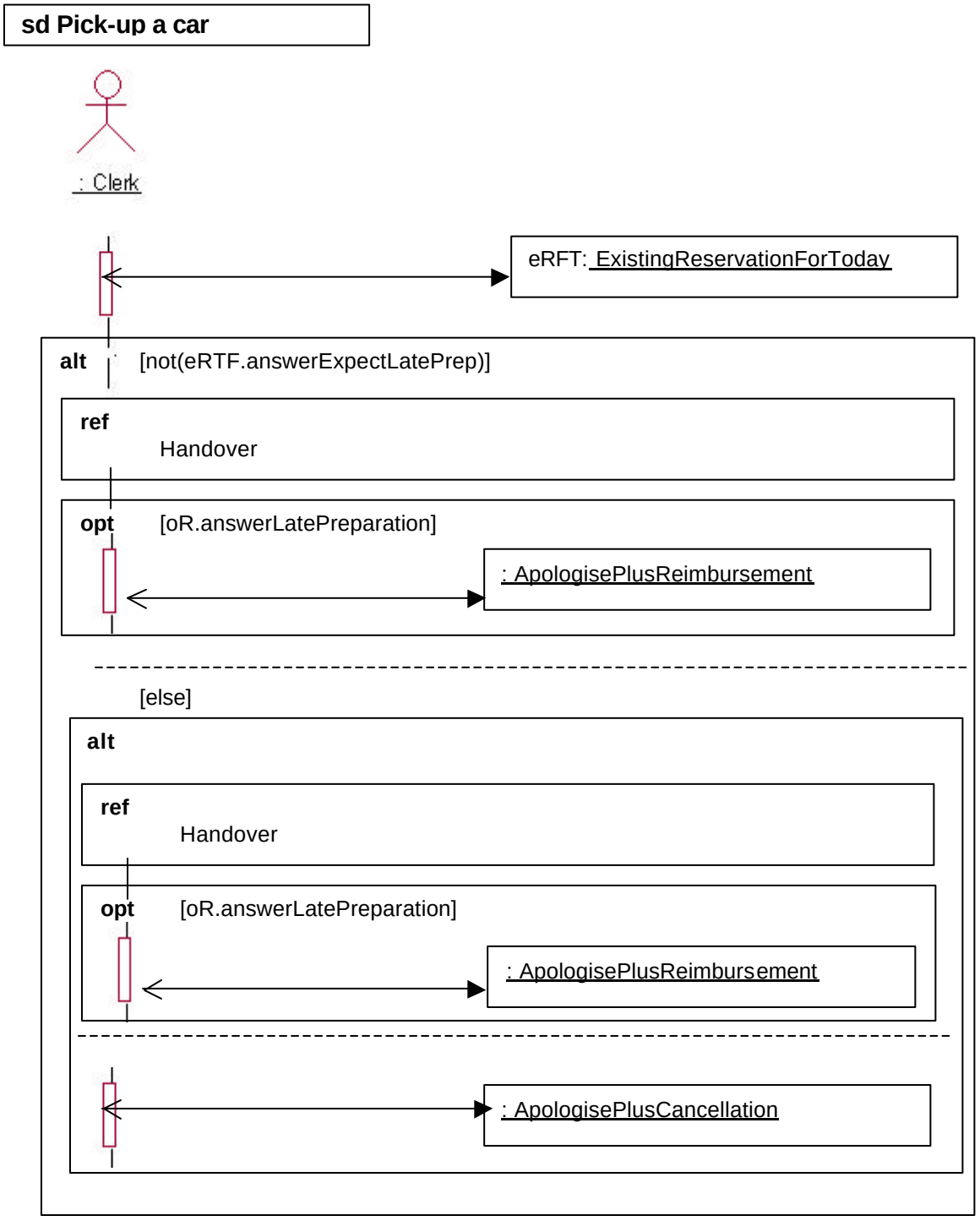




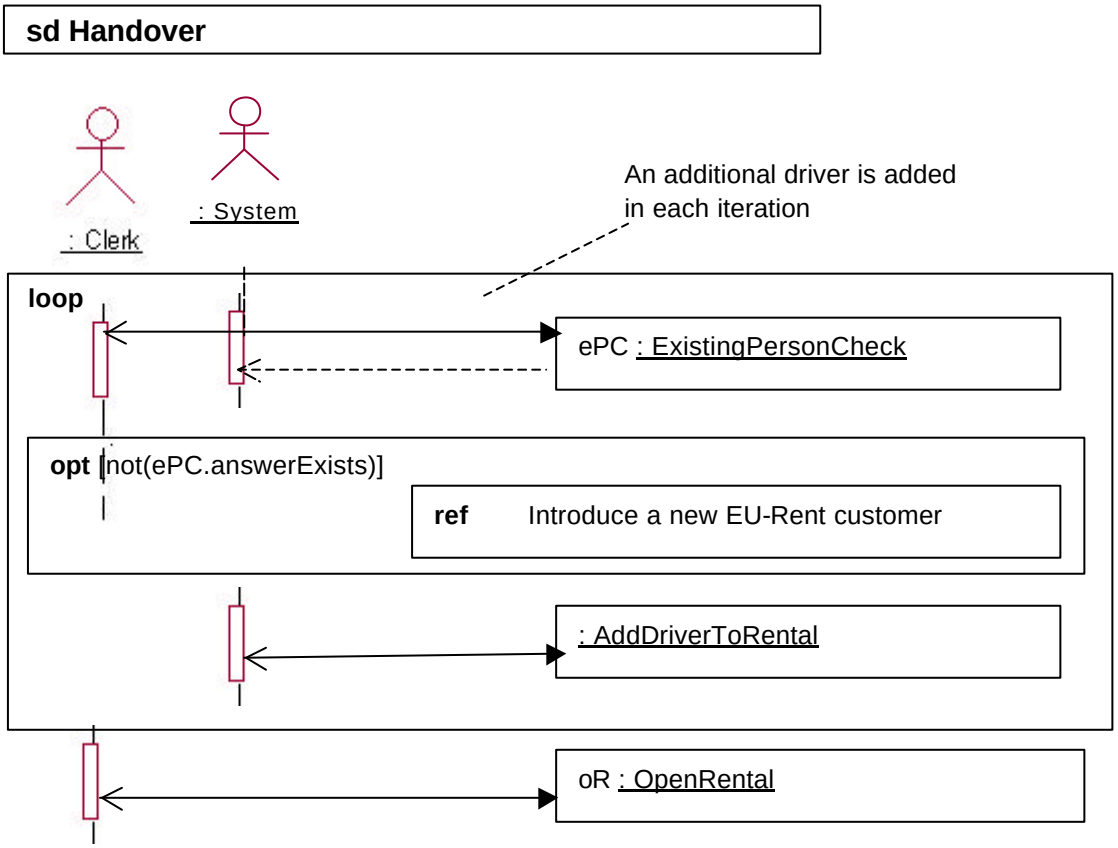

sd Get cars to be prepared
: Mechanic
: CarsToBePrepared
sd End of car preparation
: Clerk
: CarPrepared
sd End of car maintenance
: Clerk
eOM: EndOfMaintenance
opt
[eOM.answerSellCar]
ref
Sell car

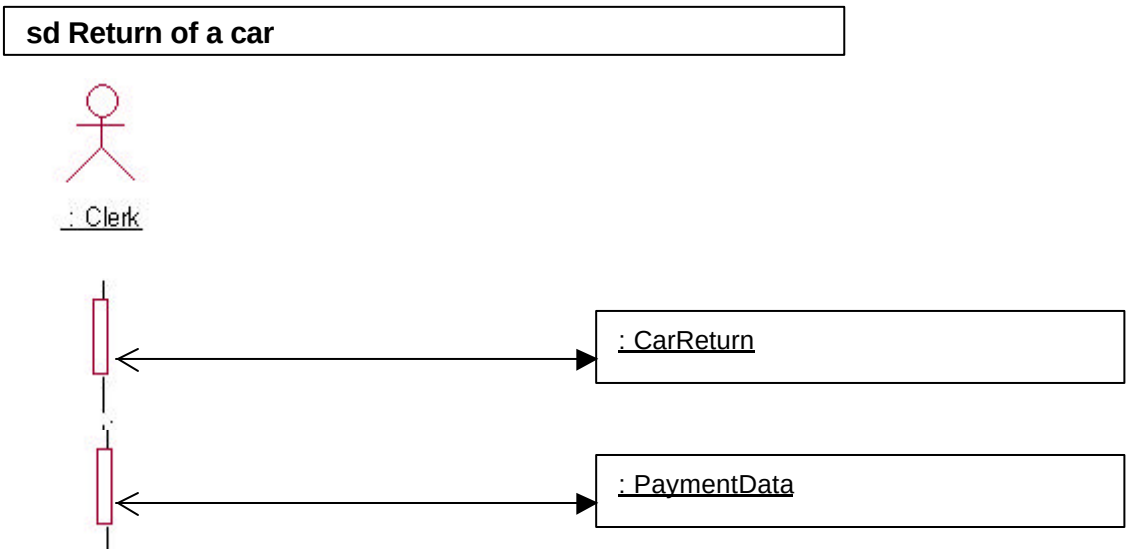

sd End of car repairs
: Clerk
eOR: EndOfRepairs
opt
[eOR.answerSellCar]
ref
Sell car

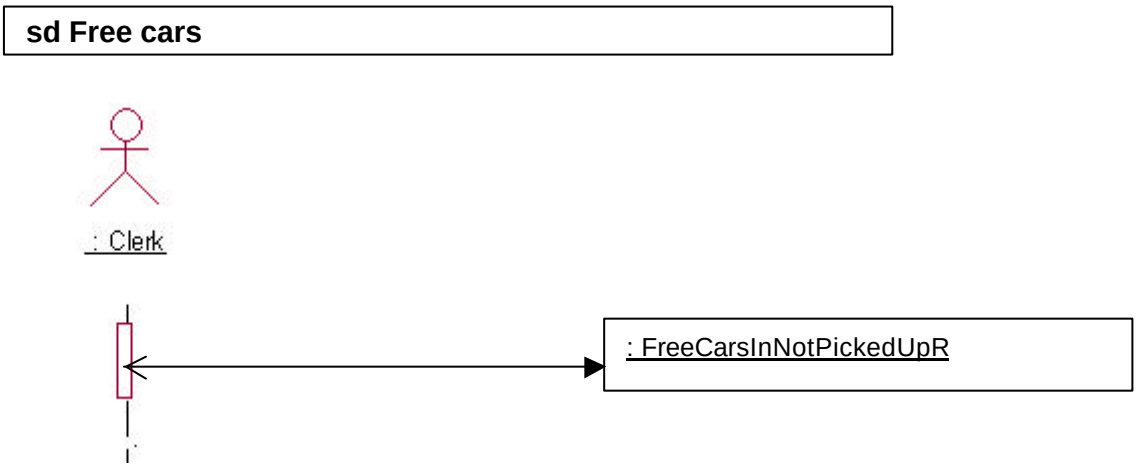
Car pick-up and return management

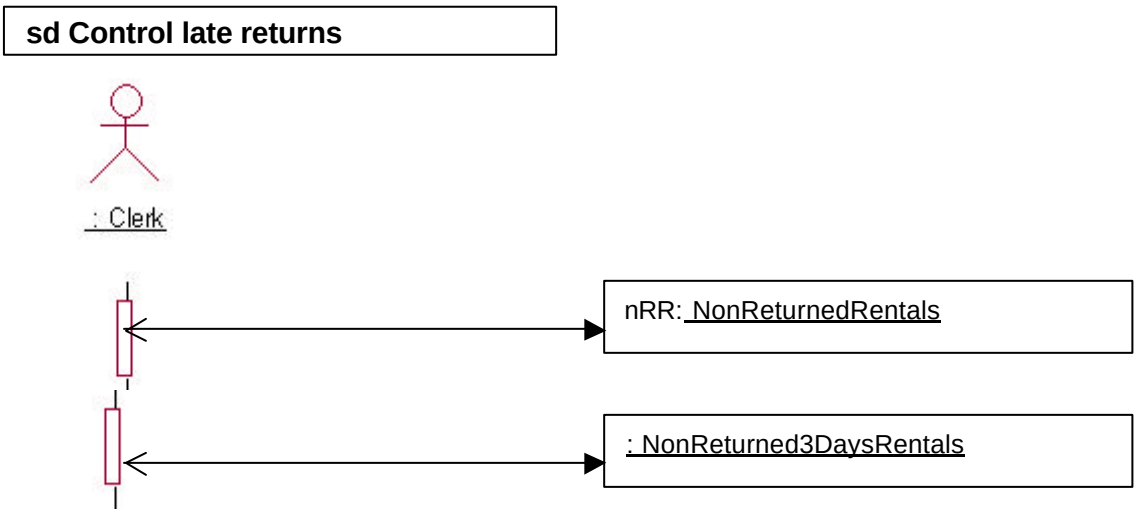


(*) Note: Handover will not begin until the car is prepared- the customer will wait-. Consequently, the following expression will be satisfied on Handover:
eRFT.answerReservation.assignedCar.oclIsKindOf(Prepared)

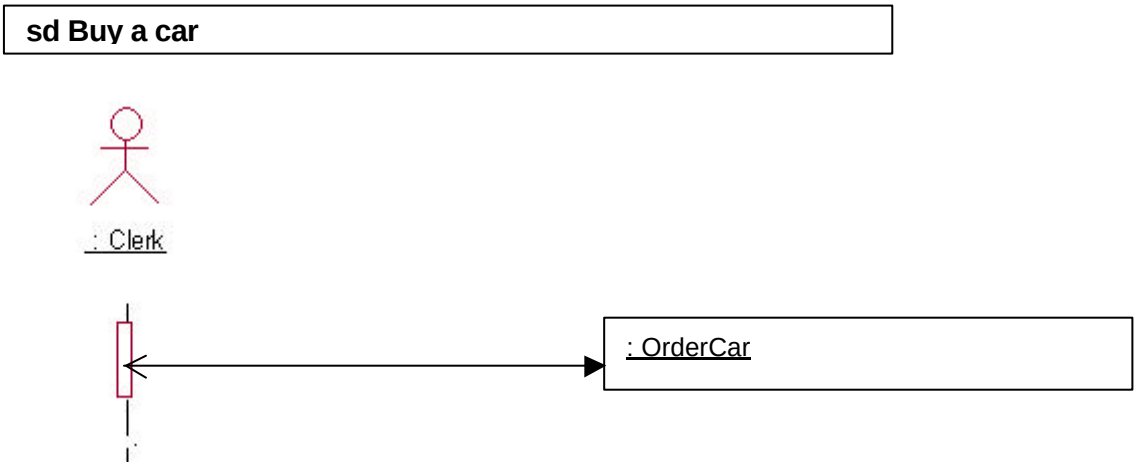

sd Handover
: Clerk
: System
An additional driver is added
in each iteration
loop
ePC : ExistingPersonCheck
opt [not(ePC.answerExists)]
ref Introduce a new EU-Rent customer
: AddDriverToRental
oR : OpenRental

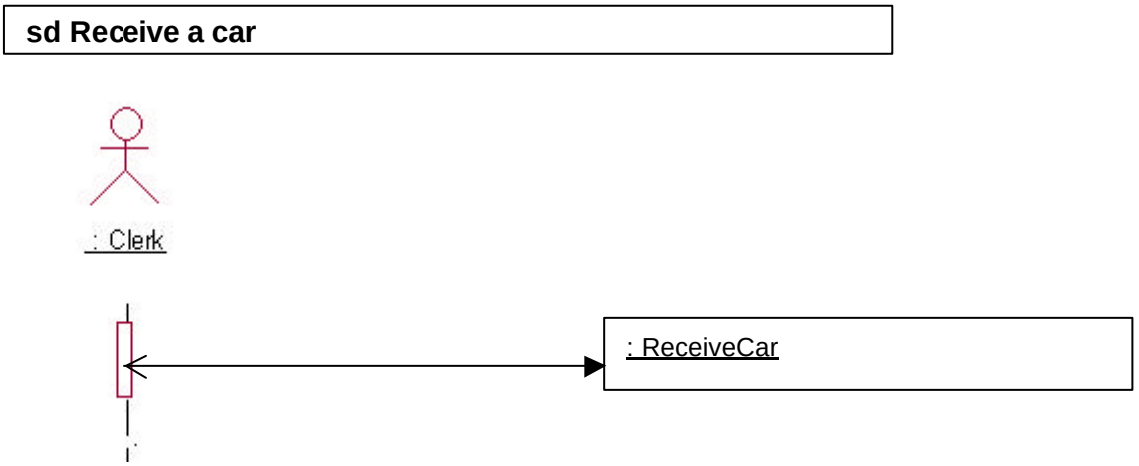

sd Return of a car
: Clerk
: CarReturn
: PaymentData

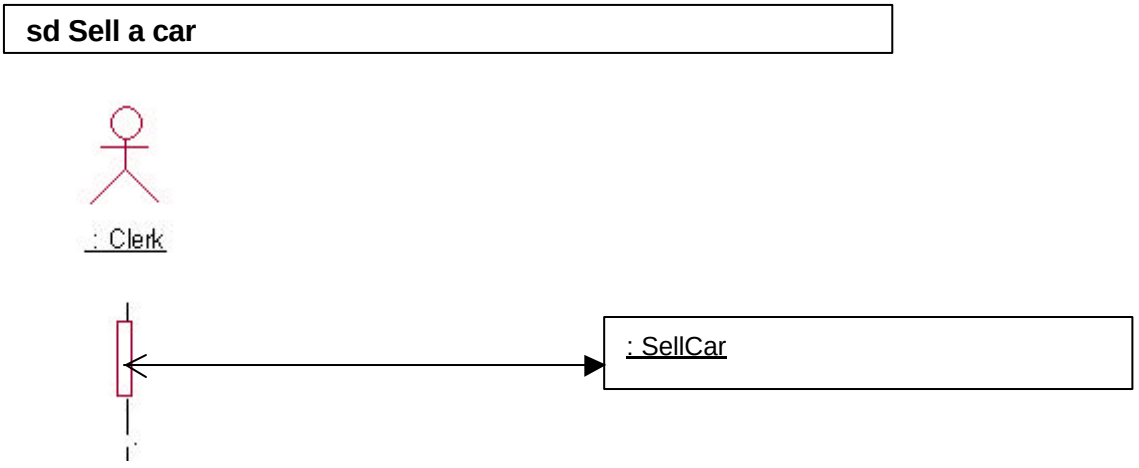

sd Free cars
: Clerk
: FreeCarsInNotPickedUpR

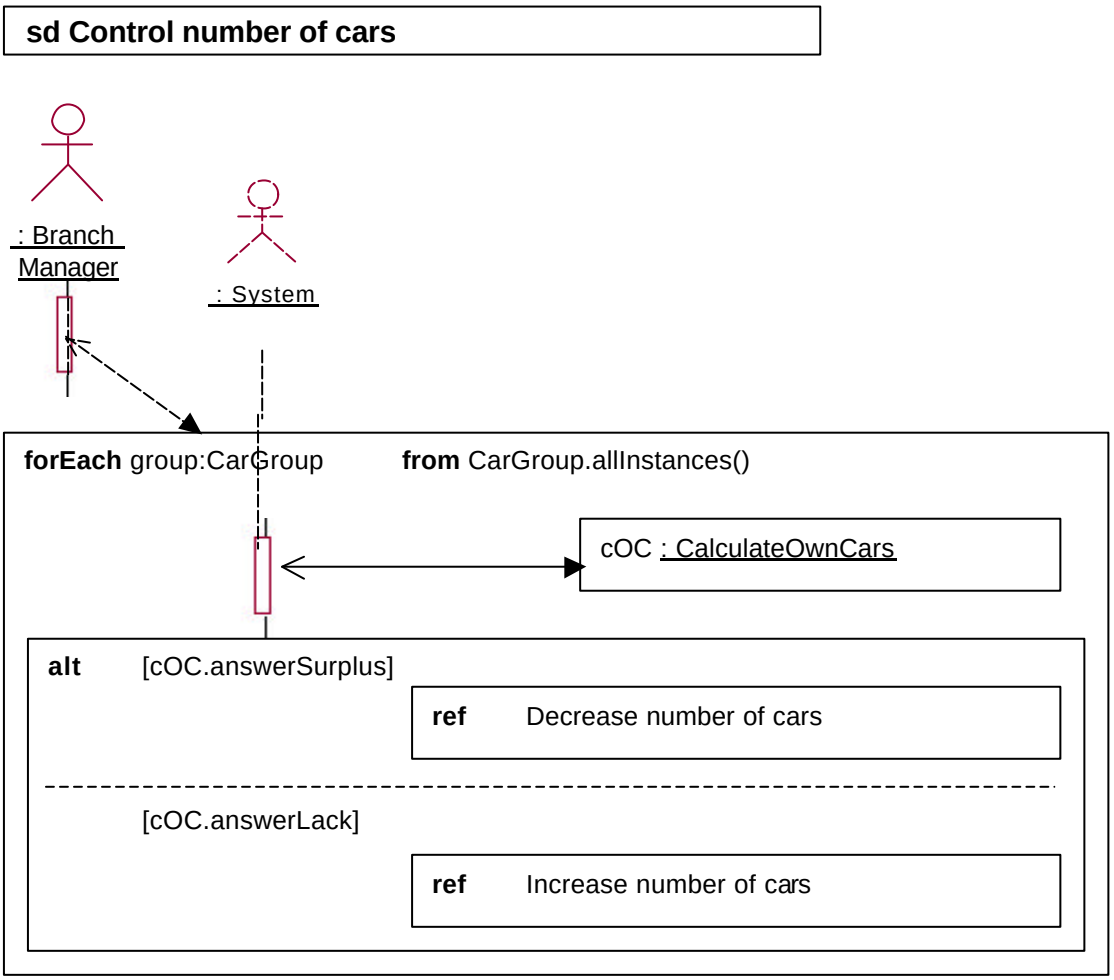

sd Control late returns
: Clerk
nRR: NonReturnedRentals
: NonReturned3DaysRentals

Car Management

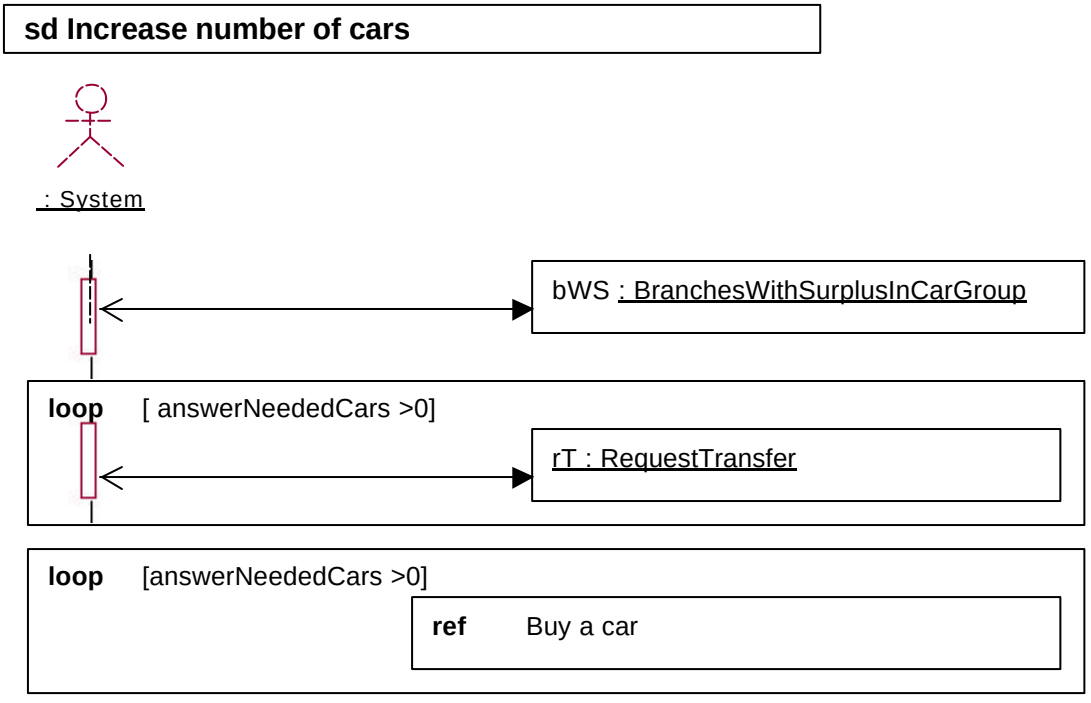


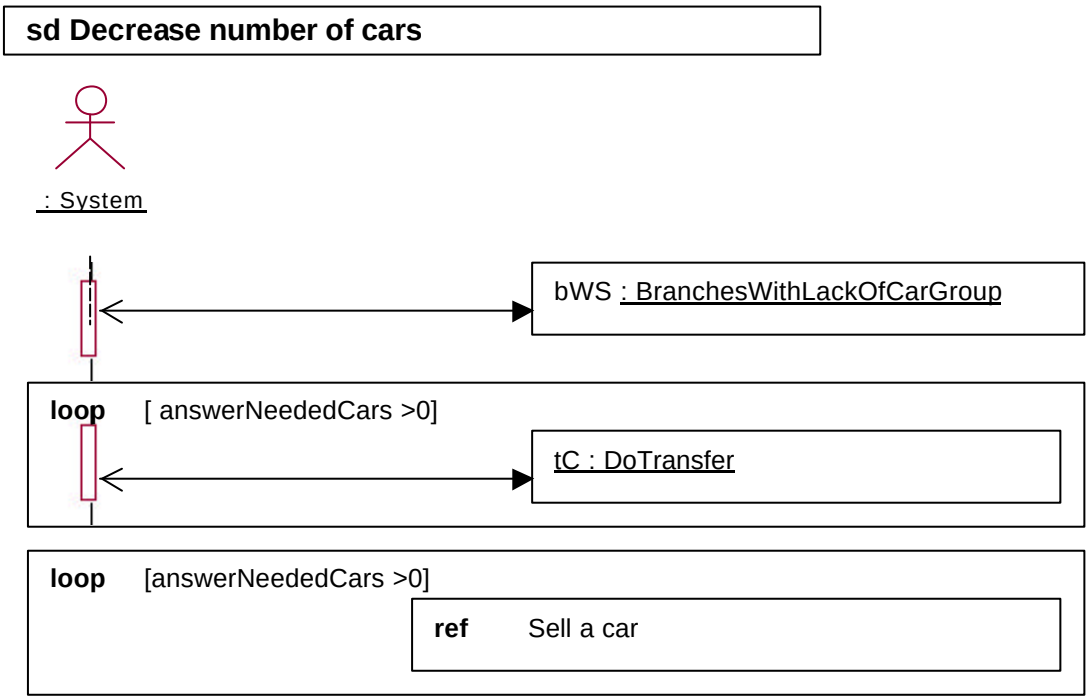




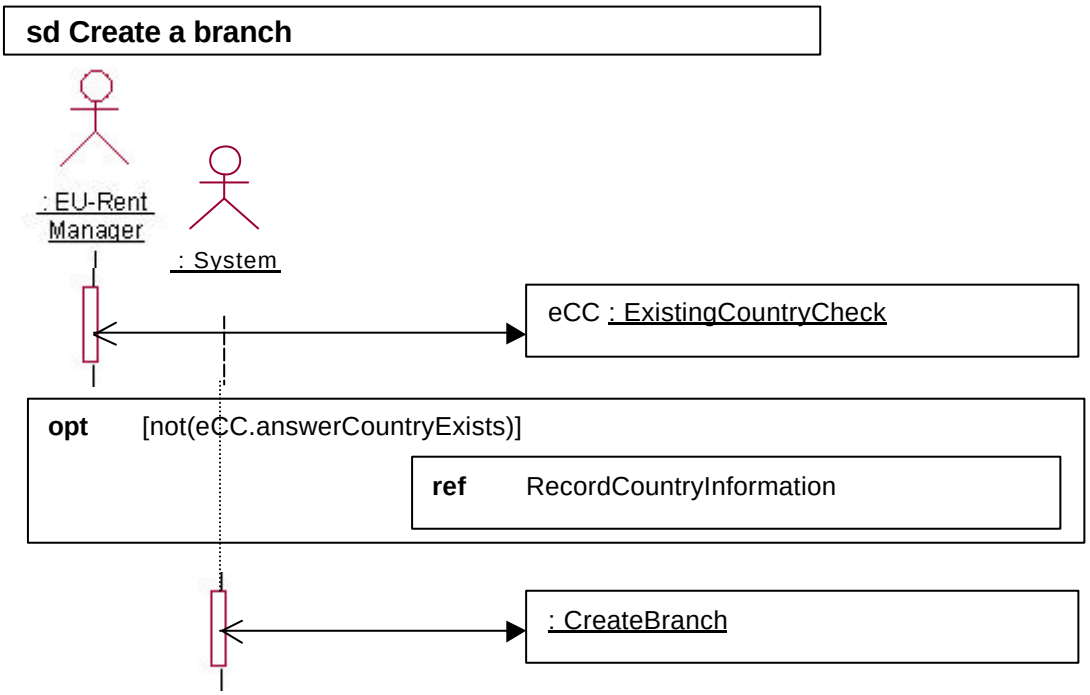


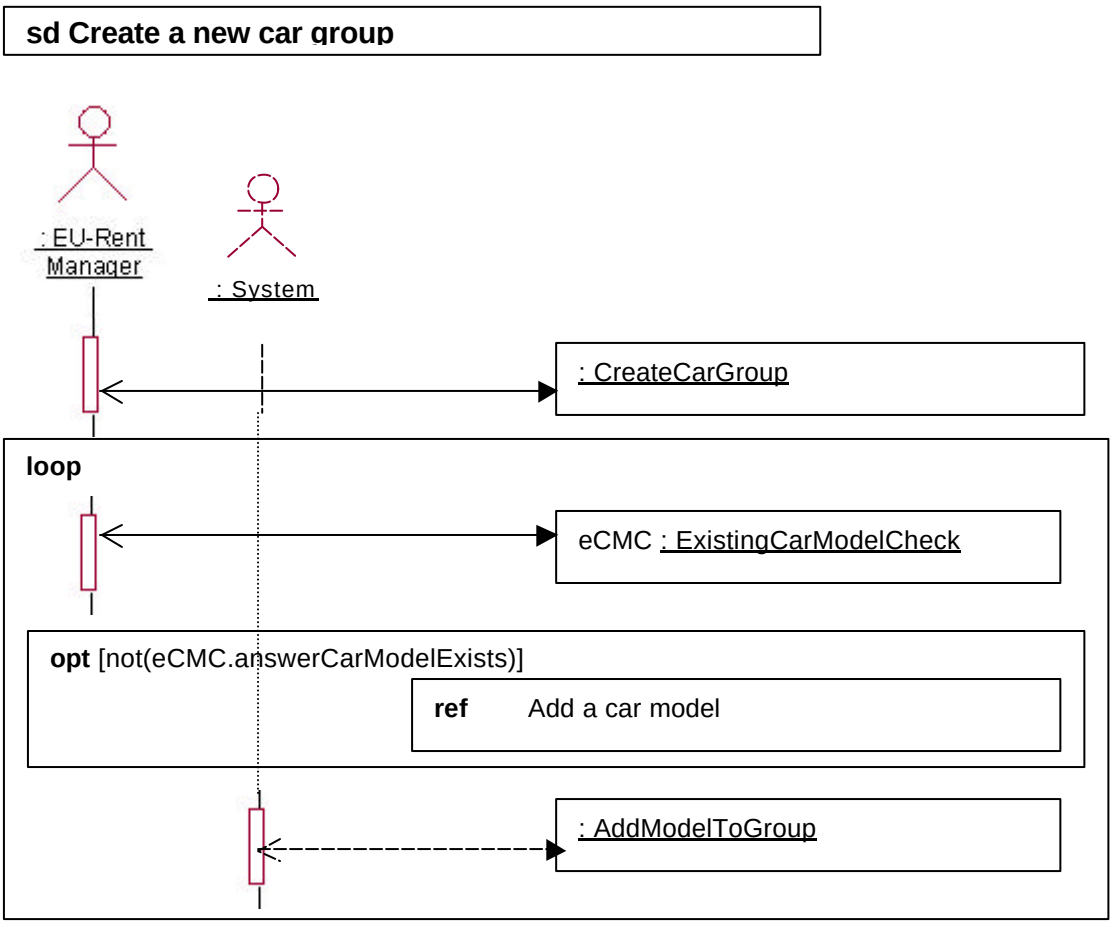
(*)Note the use of the non-standard UML 2.0 way to show that the sequence of events is sparked off by a system user (the branch manager in this case)

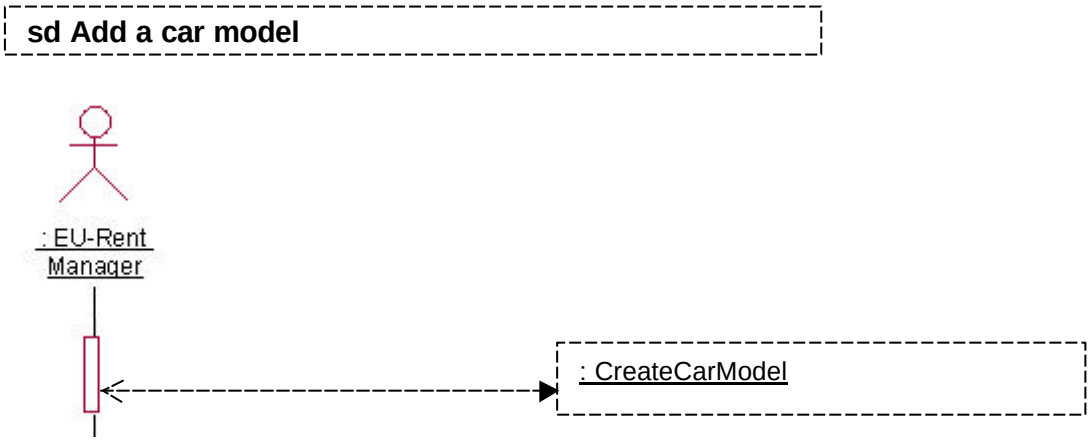


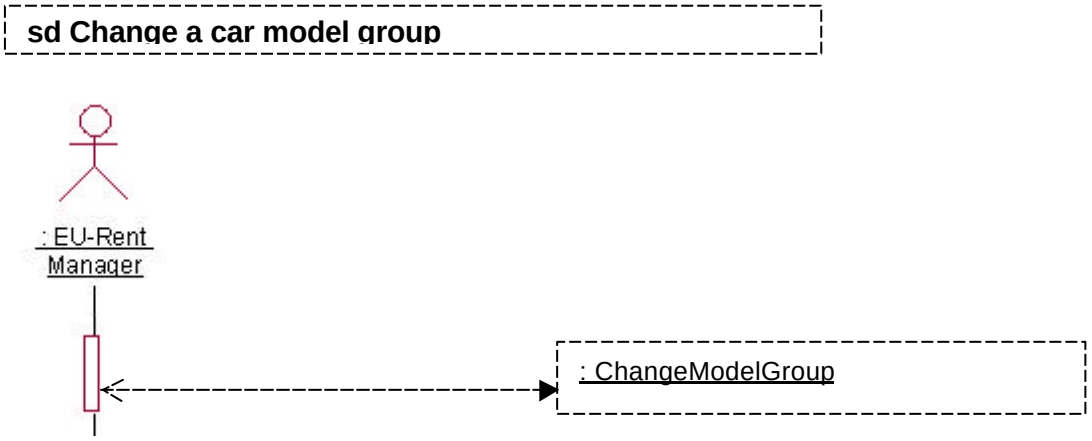

sd Decrease number of cars
: System
bWS : BranchesWithLackOfCarGroup
loop
[answerNeededCars >0]
tC : DoTransfer
loop
[answerNeededCars >0]
ref
Sell a car

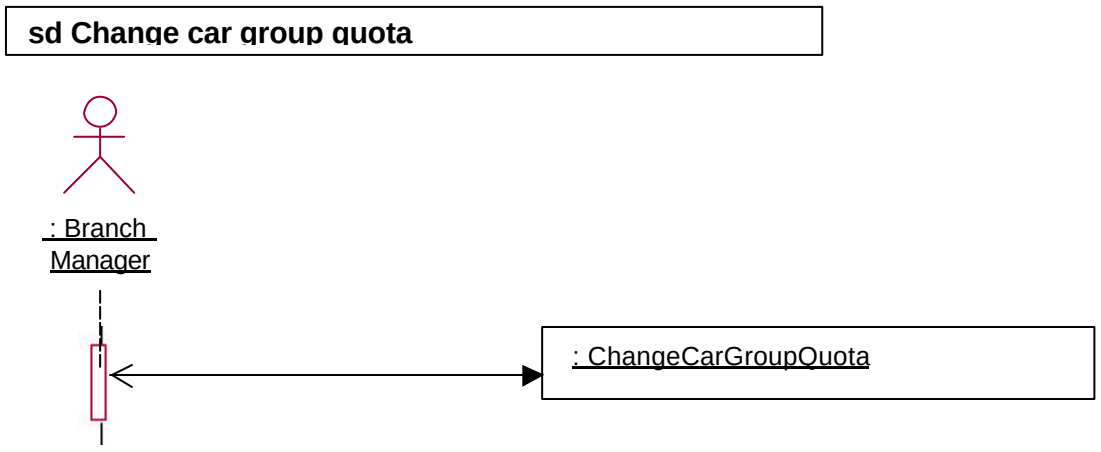
Branch, Car Group and Models management



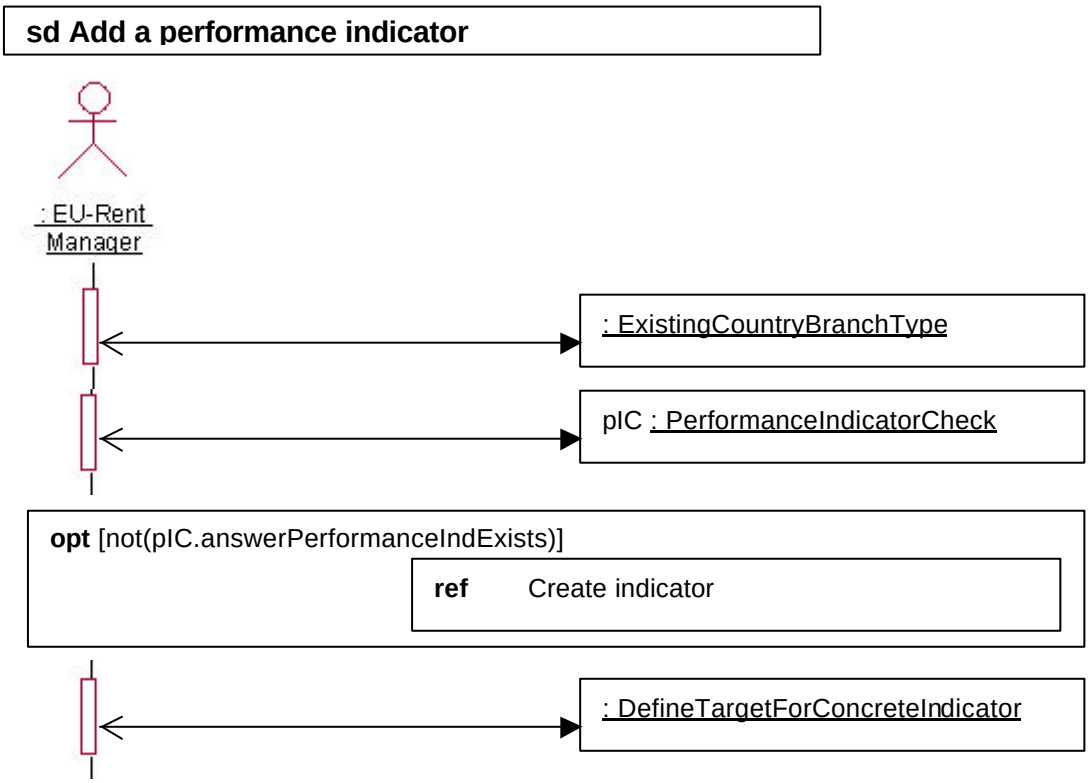


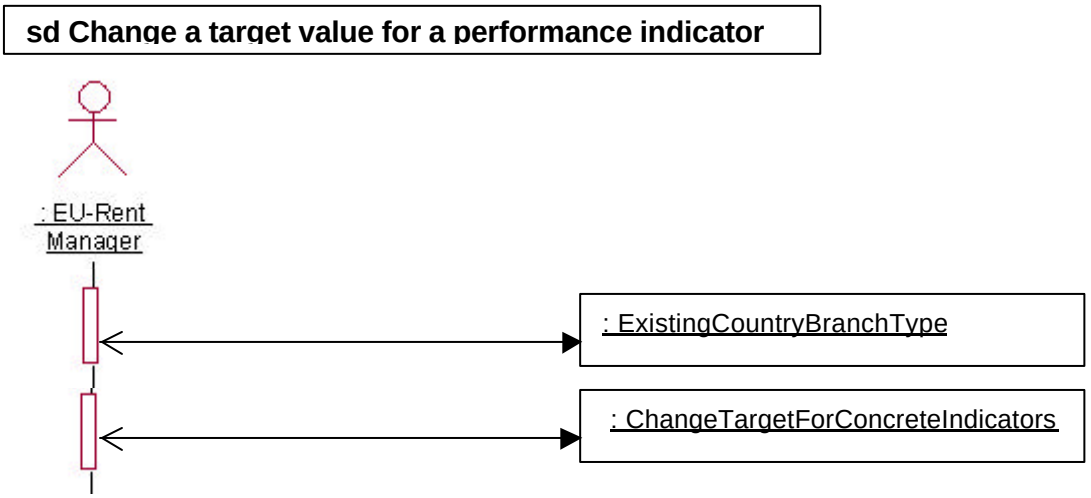

sd Add a car model
: EU-Rent Manager
: CreateCarModel


sd Change a car model group
: EU-Rent Manager
: ChangeModelGroup


sd Change car group quota
: Branch Manager
: ChangeCarGroupQuota

Performance indicators management





9. CONCLUSIONS

When starting working with the case, the first problem to be solved was to clearly define the case. However, the final version was not obtained until the use cases were written, because it was not until then that we were conscious of the real needs of the system and so, what should be clearly defined. As commented in the corresponding section, the decisions made have tried to be as consistent as possible with the original case and keep the size of the case treatable. Even so, when elaborating the specification one should be very careful to be consistent with what was stated before because the complexity of the case turned out to be considerable.

During the specification, the proposals made by Antoni Olivé in [IC-Ol03], [DR-Ol03] and [EE] were generally useful, economic and easy-to-use.

For example, the approach to define the derivation rules associated to derived elements based on operations definition, allowed the exploit of redefinition mechanism in non-trivial attributes such as the calculation of the corresponding best price of a rental agreement. Besides the mechanism was also useful for defining hybrid types such as *driver.*

An analogous approach to define integrity constraints also proved to be useful in defining integrity constraints in general, and creation time in particular, which otherwise could not have been defined.
Furthermore, these techniques proved to be also very convenient when used jointly with the proposal of modelling system events as objects, suggested in [EE].

One of the main advantages of this different approach to model events is the ease of reuse of constraints such as existence constraints, which are very common. However, the approach is also convenient to encapsulate “apply actions” (that is, postcondition elements) by a hierarchy definition. This hierarchy takes basically two forms in the project. In the first structure, the parent is abstract (or could be so) and defines some conditions or changes common to possibly several events, which inherit this object. While in the second structure, the parent is not abstract and defines some basic conditions, which a derived child with special characteristics extends (and therefore we minimize the use of conditional structure and clarify the model).

On the other hand, one difficulty found when defining the system events was how to *classify* some of them. One example is in the context of car allocation, where the system can request the user to choose among some options if an exception or an in extremis rule has to be applied. In this case, it is not the user who decides to perform an action, but the system, and so, it cannot be considered an action request event strictly; however it has been considered so in this document. The idea to do it is taking into account that the overall sequence of events, to which the event belongs, is sparked off by the user and so, all the component events are, in a sense, caused by the user. Anyway, it makes rethinking if a more accurate classification may be convenient.

Finally, it should be noticed one drawback related to derived elements. The problem is that in one system such as EU-Rent Rentals case, many convenient derived elements or which just make sense can be defined. Therefore, the representation of these elements joined with all the rest (the non-derived) can considerably enlarge the size of the model and so, make the representation unclear or excessively heavy for a human eye. This situation forces the specifier either to divide the model in coverable pieces or

alternatively, have some way (preferably offered by the modelling tool) of *hiding* them when convenient.
To sum up, we believe that this project not only has been generally successful on its original objectives but has also served to experiment with some highly topical subjects such as UML 2.0. and remark the importance of a clear diagram structure.

One possible future work line could consist on refining the event approach to be widely used in practice. This refinement should consider the inclusion of new types or subtypes of events, as well as some criteria or tips to split the events and determine split events generating actor (in fact, this is a common problem of any event modelling approach).

10. REFERENCES

- Original Case Study:
  [BRG95] *Appendix D of the paper "Defining Business Rules ~ What Are They Really?", produced by the Business Rules Group, 1995. (http://www.businessrulesgroup.org/first_paper/br01ad.htm)*

- Case Study Extensions:
  - [BRF03] *Business Rules Forum (http://www.businessrulesforum.com/derby.html).*
  - [EBRC03] *European Business Rules Conference, June 2003 (http://www.eurobizrules.org/eurent.htm)*
  - [PSZ00] *Advances in object-oriented data modelling,* M.P. Papazoglou, S. Spaccapietra, Z. Tari Ed.), Cambridge, Massachusetts, USA, MIT Press, 2000.

- Use Cases:
  - [Coc00] *Writing Effective Use Cases*. Alistair Cockburn, Octuber 2000.
  - [Wei03] *Adopting use cases. Part I: Understanding types of use cases and artifacts.* Pan-Wei Ng. The rational edge, May 2003. http://www.therationaledge.com/may_03/m_ng.jsp
  - [Gel03] *Precise Use Cases.* David Gelperin. Live Specs software. http://www.livespecs.com/modules.php?op=modload&name=News&file=index&catid=14&topic=&allstories=1&POSTNUKESID=57c8939a70d06566fd4ee4a69cbd1aac

- Specification (general):
  - [Lar02] *UML y Patrones*. Craig Larman. Prentice Hall, second edition, 2002.
  - [OMG01] *OMG. Unified Modelling Language Specification, version 1.5.* March 2003. http://www.omg.org/technology/documents/formal/uml.htm
  - [UMLS03] *Unified Modelling Language: Superstructure. Version 2.0. (3rd revised submission to OMG RFP ad/00-09-02*), April 2003.
  - [OCL03] *Response to the UML 2.0 OCL RfP (ad/2000-09-03). Revised Submission, Version 1.6, January 6, 2003. (OMG Document ad/2003-01-07).*
  - [XML01] *XML Schema Part 0: Primer. W3C Recommendation, 2 May 2001.* http://www.w3.org/TR/xmlschema-0/
  - [IC-Ol03] *Integrity Constraints Definition in Object-Oriented Conceptual Modelling Languages.* Antoni Olivé, 2003.
  - [DR-Ol03] *Derivation Rules in Object-Oriented Conceptual Modelling Languages*. Antoni Olivé, 2003.
  - [EE] *Events and their effects*.*(Not definitive version)* Antoni Olivé.

12. ACKNOWLEDGEMENTS

This work has been partially supported by the *Ministerio de Ciencia y Tecnologia* and FEDER under project TIC2002-00744.

11. SUMMARY